9-2009

Action-Based Representation Discovery in Markov Decision Processes

Sarah Osentoski
University of Massachusetts Amherst, sosentos@cs.umass.edu

ACTION-BASED REPRESENTATION DISCOVERY IN MARKOV DECISION PROCESSES

A Dissertation Presented

by

SARAH OSENTOSKI

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2009

Department of Computer Science

ACTION-BASED REPRESENTATION DISCOVERY IN MARKOV DECISION PROCESSES

A Dissertation Presented

by

SARAH OSENTOSKI

Approved as to style and content by:

_____

Sridhar Mahadevan, Chair

_____

Andrew G. Barto, Member

_____

Roderic Grupen, Member

_____

Andrea Nahmod, Member

_____

Andrew G. Barto, Department Chair
Department of Computer Science

To my grandmothers: Doris Barrons and Theresa Osentoski.

ACKNOWLEDGMENTS

I start by thanking my advisor Sridhar Mahadevan. His guidance, advice, and support were instrumental in shaping this research and guiding my development as a researcher. Working with Sridhar taught me how to identify interesting problems and how to study them using many different tools.

I am also grateful to the other members of my committee for their support and guidance. I have been very privileged to have many interactions with Andrew Barto as part of the Autonomous Learning Lab. Andy has been a great source of inspiration and a wonderful example. His suggestions and criticisms have made me a more careful and thorough researcher. I have also been privileged to collaborate with Roderic Grupen during my time at UMass. Rod is an outstanding researcher who helped me look at problems from a broader perspective. I would like to thank him for his constant support, his faith in my work, and for all of the time he spent discussing ideas with me. I thank Andrea Nahmod for her insightful comments and advice.

Many others have contributed to the development of the ideas in this dissertation and to my development as a researcher. I thank George Konidaris, Steve Hart, Alicia "Pippin" Wolfe, Jeff Johns, and Ashvin Shah for their insightful conversations and camaraderie. I thank Russell Duhon for the many conversations about the broader perspective and impact of research.

I am grateful to the members, past and present, of the Autonomous Learning Laboratory and the Laboratory for Perceptual Robotics for the many helpful discussions, support and friendship. Thanks to Mohammad Ghavamzadeh, Mike Rosenstein, Anders Jonsson, Özgür Şimşek, Kimberly Ferguson, Alicia P. Wolfe, Rob Platt, Balarman Ravindran, Victoria Manfredi, Steve Hart, Aron Culotta, Andy Fagg, Colin Barringer, Jeffrey Johns, Chang

Wang, George Konidaris, Suchi Saria, Andrew Stout, Chris Vigorito, TJ Brunette, Khashayar Rohanimanesh, Shriaj Sen, Ashvin Shah, Bruno Castro da Silva, William Dabney, Scott Kuindersma, and Vimal Mathew.

I could have never made it through the graduate program without the support staff at UMass. I thank Leeanne Leclerc and Sharon Mallory for helping me navigate my way through the graduate program. I also thank Laurie Downey and Gwyn Mitchell for their help with my numerous questions over the years.

I thank all of the wonderful friends I have made in Amherst. The fun times we had made grad school more enjoyable. I thank Emily Horrell, Kimberly Ferguson, Lisa Friedland, Ilene Magpiong, Audrey Lee, Victoria Manfredi, Maryanne Olson, Emily Russo, Özgür Şimşek, and Hanna Wallach for the wonderful dinners we shared and their friendship. I also thank Stephen Murtagh and Gene Novark for their friendship and the wonderful conversations shared over a few pints.

I thank my family for their love and encouragement. My parents have been very supportive of me in all my endeavors. I also thank my siblings for the crazy conversations and entertaining stories. My family always challenges to me think about my priorities and keeps me honest.

Last I would like to thank Andrew Cosand for his constant support and unwavering faith as I finished my dissertation. Our shared conversations and adventures have made completing this dissertation a more enjoyable process. Thank you, Andy.

ABSTRACT

ACTION-BASED REPRESENTATION DISCOVERY IN MARKOV DECISION PROCESSES

SEPTEMBER 2009

SARAH OSENTOSKI

B.Sc., UNIVERSITY OF NEBRASKA LINCOLN

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Sridhar Mahadevan

This dissertation investigates the problem of representation discovery in discrete Markov decision processes, namely how agents can simultaneously learn representation and optimal control. Previous work on function approximation techniques for MDPs largely employed hand-engineered basis functions. In this dissertation, we explore approaches to automatically construct these basis functions and demonstrate that automatically constructed basis functions significantly outperform more traditional, hand-engineered approaches.

We specifically examine two problems: how to automatically build representations for action-value functions by explicitly incorporating actions into a representation, and how representations can be automatically constructed by exploiting a pre-specified task hierarchy. We first introduce a technique for learning basis functions directly in state-action space. The approach constructs basis functions using spectral analysis of a state-action

graph which captures the underlying structure of the state-action space of the MDP. We describe two approaches to constructing these graphs and evaluate the approach on MDPs with discrete state and action spaces.

We show how our approach can be used to approximate state-action value functions when the agent has access to macro-actions: actions that take more than one time step and have predefined policies. We describe how the state-action graphs can be modified to incorporate information about the macro-actions and experimentally evaluate this approach for SMDPs with discrete state and action spaces.

Finally, we describe how hierarchical reinforcement learning can be used to scale up automatic basis function construction. We extend automatic basis function construction techniques to multi-level task hierarchies and describe how basis function construction can exploit the value function decomposition given by a fixed task hierarchy. We demonstrate that combining task hierarchies with automatic basis function construction allows basis function techniques to scale to larger problems and leads to a significant speed-up in learning.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

A hallmark of human intelligence is the ability to adapt to new environments and to learn new tasks. These abilities are also desirable in autonomous agents. For example, a robot may need to navigate in a new environment or to manipulate new objects. Such sequential decision problems involve significant uncertainty, and are often modeled as Markov decision processes (MDPs) (Puterman, 1994). An MDP is a mathematical model that represents a problem as a set of states S, a set of actions A, a stochastic transition distribution P that describes the outcome of selecting action a in state s and a reward function R. The agent selects actions that change its environment and then selects new actions based upon feedback from the environment, such as a reward signal or changes in the environment. The objective of the agent is to learn a *policy* or a mapping from states to actions that maximizes its long-term cumulative discounted reward.

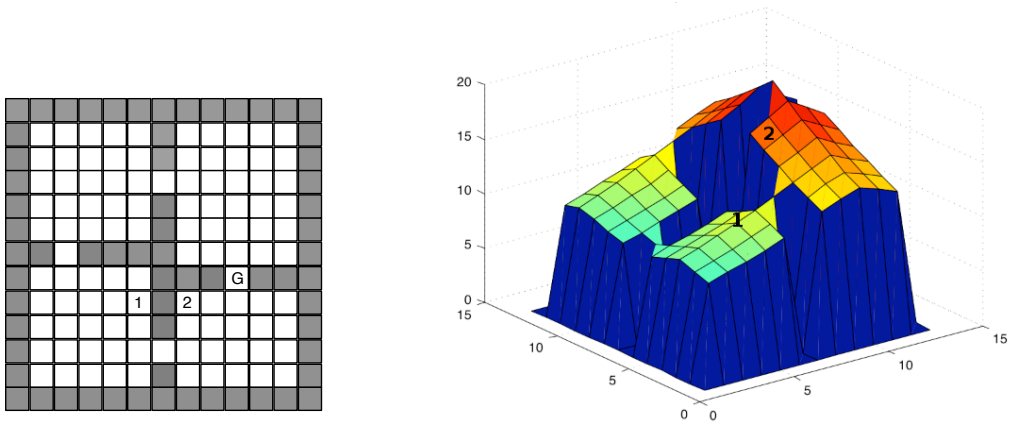
The agent must learn which action are responsible for the outcome of the task, given that he agent may not be rewarded frequently, and that many actions may be taken before reward is given. One way the agent can solve this problem is to construct a value function over the state space where the value for a state is the expected sum of immediate reward received and the expected value of the next state. There are many approaches to solving Markov decision processes, such as linear programming (de Farias & Van Roy, 2003), policy iteration (Howard, 1960), and value iteration (Puterman, 1994). In this dissertation, we focus on reinforcement learning (RL), a machine learning paradigm designed learn the best action given an agent's experience in the domain. RL algorithms, such as Q-learning

(Watkins, 1989), can be used to approximately solve MDPs, by propagating values back across states and actions as the agent gains more experience in the environment.

Function approximation techniques are necessary when exact representations become infeasible, such as a large or continuous state space. These approaches are also valuable in accelerating the convergence of many RL algorithms. Samuel (1959) introduced one of the first examples of function approximation in the game of checkers. In Samuel's paradigm, agents do not learn the underlying representation but instead leverage this representation to learn policies. Amarel (1968) introduced a paradigm where agents learned representations built through global analysis of the state space. Most previous work in approximately solving large MDPs used a set of hand engineered features, also known as *basis functions*. Basis functions map a state $s \in S$ to a k-dimensional real valued vector $\phi(s)$ where $k \ll |S|$. Examples of popular basis functions are radial basis functions (RBFs) (Powell, 1987; Sutton & Barto, 1998; Lagoudakis & Parr, 2003), "cerebellar model articulator controllers" (CMACs) (Albus, 1981; Watkins, 1989; Sutton & Barto, 1998), polynomials (Bellman & Dreyfus, 1959; Lagoudakis & Parr, 2003), and neural networks (Farley & Clark, 1954). A linear combination of basis functions is used to represent the value function $V = \Phi\boldsymbol{\theta}$, where $\boldsymbol{\theta}$ is the parameter vector and Φ is a matrix whose columns are the basis functions. Φ provides a low-dimensional projection of the original value function in $\mathbb{R}^{|S|}$ onto a smaller subspace in $\mathbb{R}^k$. Φ also induces a reduced MDP and can be used to compress any function over the MDP, not just value functions (Mahadevan, 2009).

Previous approaches typically hand-engineered basis functions, creating a low dimensional subspace. However, the success of these approaches depend upon the designer creating appropriate basis functions. These approaches often assume that the underlying space has Euclidean geometry, but states that are close in Euclidean space may have values that are far apart (Dayan, 1993; Drummond, 2002). Figure 1.1 illustrates a four room grid-world in which two states are labeled. Figure 1.1(b) illustrates that while these states are close in Euclidean space, they have very different values. Additionally, traditional function

approximation approaches did not solve the problem of automatically constructing basis functions.



(a) Four room gridworld (b) Example value function for the four room gridworld

Figure 1.1. Four room gridworld domain an example of a value function.

In this dissertation we propose a paradigm in which an agent automatically discovers representations and uses these representations for learning policies. This dissertation specifically focuses on automatically constructing a set of compact low-dimensional basis functions Φ to represent an MDP that will aid the agent in efficiently solving the MDP.

Definition 1.1 Automatic Basis Construction Problem: *Given a Markov Decision Process $M = (S, A, P, R)$, automatically construct a low-dimensional representation Φ where the size of Φ is $|\tilde{S}| \times k$ or $|\tilde{S}||A| \times k$ where $\tilde{S} \subset S$ and $k \ll |\tilde{S}|$ or $k \ll |\tilde{S}||A|$. Φ should be constructed such that the solution of M calculated using Φ closely approximates the solution of the original MDP M.*

There are several considerations that must be addressed when selecting a basis function construction technique. The first is the information available to the agent. Some approaches require the full model of the MDP (Poupart & Boutilier, 2002; Parr et al., 2007), others require a set of samples from the domain, or an assumption about distance metrics over the state space. The second is the cost of constructing the basis functions. Typically we would

like the cost of constructing the basis to be less than the cost of solving the MDP; however, this is not possible for all approaches. These approaches can be justified if the agent must solve multiple similar MDPs. The complexity of the basis is another consideration. For large sparse problems it may be desirable to have a basis that is also sparse. Another important consideration concerns whether the basis functions should be *reward-sensitive* or *reward-independent*. Reward-sensitive bases incorporate information about the reward, while reward-independent bases do not. Reward-sensitive bases are often more effective at compressing the value function, but reward-independent bases are more reusable in similar MDPs that have different value functions. Bases can also be constructed for a specific policy. Policy-specific bases will be useful for a short period of time during learning, but may be more effective at compressing the value function under the policy. The last consideration we will mention is incremental versus batch methods. Incremental approaches build representations incrementally as the agent learns while batch methods construct multiple, or all, basis functions at a time.

In this dissertation, we specifically examine approaches to automatically constructing basis functions that are jointly defined over states and actions. In particular, the aim of this dissertation is to construct basis functions that can approximate any function $f(s, a)$ over states $s \in S$ and actions $a \in A$. One example is the action-value function $Q(s, a)$. Most traditional function approximation approaches solve the basis construction problem exclusively over the state space or build separate function approximators for the state and action spaces. However, the ability to generalize across states and actions simultaneously is a skill that comes easily to humans. Consider a baker as an illustrative example. The baker could bake a variety of items but is focused upon bread and cookies. There are also a variety of actions that he will employ during the baking process, two of which are adding butter and adding yeast. If the baker were to generalize over state alone, the bread and cookies might seem initially quite similar since their primary ingredient is flour. If he were to generalize over the actions, adding yeast and adding butter would seem quite different.

However, the baker actually generalizes across both states and actions. When considering bread, state-action pairs for yeast and butter will be quite similar as they are necessary for a successful product. When considering cookies, these two actions are quite different since adding yeast will take the process to a disastrous section of the state space while butter is necessary.

Specifically we examine two problems: is it possible to automatically build representations for action-value functions by explicitly incorporating actions into the representation and can representations be automatically constructed for hierarchical reinforcement learning problems in a way that takes advantage of the action hierarchy?

Many RL algorithms compute an action-value function, which is used to derive the policy. Action-value functions explicitly represent the value of a state-action pair. $Q(s, a)$, an example of an action-value function, gives the value of the agent being in state s and selecting action a. Function approximation is necessary if the space required to store the value function is too large or if generalization is desired. We explore a technique that automatically builds representations in state-action space for action-value function approximation. However, these representations could be used to represent any arbitrary function over state-action space, such as a policy. We explore how to build representations in this space and how using representations built in this space affects learning.

A significant advance in RL has been the introduction of temporal abstraction frameworks and hierarchical learning algorithms (Barto & Mahadevan, 2003). These frameworks allow the agent to employ temporally-extended actions that allow it to make decisions at different time scales. We explore how to build representations that incorporate information about the temporally-extended actions including how to leverage a task hierarchy when one is available.

1.1 Overview of Approach

Our approach to building compact representations in state-action space is based on recent work that constructs basis functions on graphs induced by an MDP. This approach is convenient for two reasons: incorporating actions into the framework is straightforward and the approach captures the underlying structure of the domain. Specifically we build upon the graph Laplacian eigenfunction approach to building basis functions (Mahadevan, 2005). However, our approach can easily be extended to other approaches such as wavelets (Maggioni & Mahadevan, 2006a) or geodesic Gaussian kernels (Sugiyama et al., 2007).

This approach introduces a novel type of function approximation by deriving the bases from the topology of the underlying state space graph. Many RL algorithms learn a value function and then derive a policy from the value function. The goal of these techniques is to represent value functions compactly. To do this, function approximation techniques build a reduced representation in a feature space. A desirable feature space is one in which similar state-action pairs (or states) are mapped to similar points in feature space. Function approximation has two primary benefits: generalization and compact representations. Our approach is specifically useful for situations where the agent wishes to approximate an action-value function.

Linear function approximation techniques map each state-action pair (s, a) (or state) into a feature vector $\phi(s, a)$. These basis functions can be used to approximate any function defined on the state-action space. The graph Laplacian eigenbasis is an approach to building these basis functions. The graph Laplacian eigenbasis approach consists of three steps: forming a graph from the agent's experience in the domain, calculating the Laplacian on the graph, and computing the k smoothest eigenvectors of the Laplacian. To illustrate this approach consider the simple grid world example shown in Figure 3.6. The agent begins in a domain, collects samples according to some initial policy, and builds a graph from the samples. The agent then calculates the eigenvectors of the graph Laplacian. Figure 1.2 shows the second eigenvector of the graph Laplacian. The k smallest eigenvectors can

then be used as basis functions to learn the optimal policy using a learning algorithm of the programmer's choice.

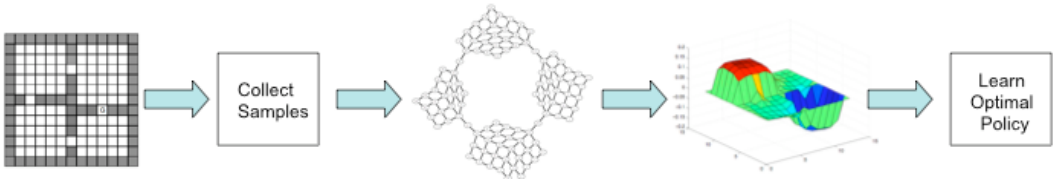


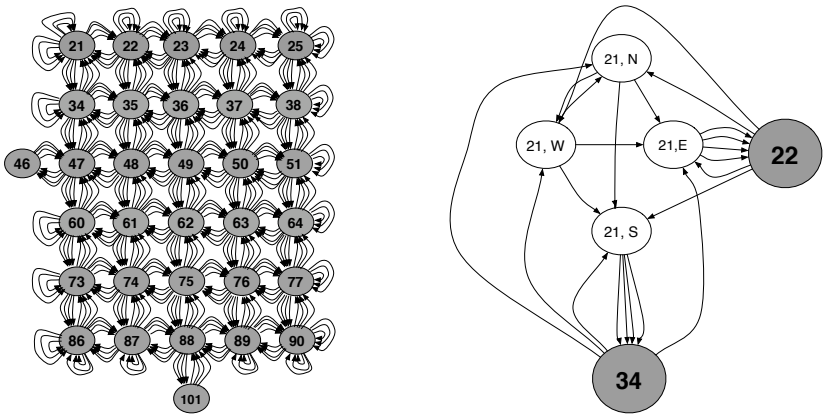
Figure 1.2. An illustration of the general approach in which the agent starts in a domain, collects samples via exploration, builds a graph, calculates the k smallest eigenvectors of the graph Laplacian and uses the eigenvectors as basis functions to represent the value function during learning.

The technique described by Mahadevan and Maggioni (2007) builds a graph over the state space and uses the eigenvectors of the graph Laplacian as basis functions. Basis functions over state-action pairs are then created by copying these basis functions for each action, zeroing out the bits corresponding to actions that were not performed. This means that the basis functions for every action in a state will be identical.

Instead, we directly represent the state-action graph, which allows the representation to vary for different actions in a state. Previous work has primarily focused on building basis functions exclusively on the state space (Lagoudakis & Parr, 2003; Mahadevan, 2005) or building separate function approximators for the state and action spaces (Smith, 2002). In order to transform basis functions built exclusively on the state space, most approaches typically copy the basis functions for each action (Lagoudakis & Parr, 2003; Mahadevan, 2005). Our approach creates fewer basis functions because it does not require saving basis functions that are copied or the extra weights for these basis functions. This is especially important in domains with a large number of actions and domains where the number of actions available in each state varies significantly; the basis functions for a state must be copied for all possible actions, even those not available in the state. Embeddings created using state-action graphs are also able to differentiate between actions when several actions

with different costs lead from state s to state s'. In state graphs these differences cannot be modeled and would be averaged or totally ignored.

In order to understand what these state-action graphs look like, consider a smaller example of the four room grid world shown in Figure 1.2. In this domain, the agent can select from four actions: north, east, south, and west. Figure 1.3 shows the state-action graph for the upper right-hand room. Figure 1.3(a) shows the global topology of the graph. Each node in this figure represents the four state-action pairs for each state. Figure 1.3(b) shows the state-action pairs for state 21. In this dissertation, we will describe an approach that allows an agent to create these graphs automatically and will demonstrate their usefulness in learning.



(a) State-action graph for the four room gridworld.

(b) Close up of transitions associated with nodes for state action pairs for state 1.

Figure 1.3. The state action graphs created for a small room.

When exploring basis function construction for hierarchical reinforcement learning, we first extend the state-action graph approach to incorporate temporally extended actions. We then examine building basis functions for multi-level task hierarchies. We introduce an approach that constructs basis functions that decompose with the task hierarchy. This approach automatically constructs basis functions for parent subtasks using basis functions from the children of a subtask. A reduced graph is created and basis functions specific to

the subtask are generated from this graph. Basis functions are built recursively from child subtasks.

1.2 Contributions

There are three contributions in this dissertation.

1. The first contribution is to introduce a technique for learning basis functions *directly* in state-action space. Function approximators built on state-action spaces are more efficient since they can capture similarities and distinctions in state-action space and do not require copying. We empirically demonstrate that function approximation using state-action graphs leads to faster learning.

2. The second contribution is to show how this technique can allow us to approximate state-action value functions when the agent has access to macro-actions: actions that take more than one time step and have predefined policies. A macro-action strings together a set of primitive actions according to its policy. When the agent has access to macro-actions, the effect of selecting different actions can lead to significantly different resulting next states. Thus, actions have significantly different effects and variable durations.

3. The third contribution is to examine how a task hierarchy can be used to scale up automatic basis function construction. One of the benefits of macro-actions is that they help the agent structure its environment. Our research shows that task hierarchies can be used to scale automatic basis function construction to large tasks. Additionally, the use of automatically constructed representations significantly improves the learning performance in hierarchical reinforcement learning problems.

1.3 Outline

The remainder of this dissertation is organized as follows. Chapter 2 provides an overview of the background material and related work for this dissertation. Chapter 3 reviews spectral bases of graphs and their use in reinforcement learning. Chapter 4 introduces state-action graphs and demonstrates how bases created from these graphs will accelerate learning. Chapter 5 extends this approach to build basis functions for function approximation in SMDPs. Chapter 6 introduces an approach to build basis functions for multi-level task hierarchies.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter introduces the mathematical framework and definitions that underly the rest of the dissertation. We specifically review Markov Decision Processes (MDPs), reinforcement learning, and value function approximation. We also review related work on representation discovery including manifold learning, basis function construction, and state abstraction.

2.1 Markov Decision Processes

In this dissertation, we use finite Markov decision processes (MDPs) (Puterman, 1994) as our framework for sequential decision making. An MDP is defined as a tuple $M = (S, A, P, R)$ where S is the set of states, and A is the set of actions. P is the transition model where $P^a_{ss'}$ specifies the probability of transitioning from state s to s' after action a is taken. R is the reward function: $R^a_{ss'}$ is the reward for taking action a in state s and transitioning to s'. We will denote the set of actions admissible for a state s as $A(s)$.

The agent can compute a policy π where $\pi(s, a)$ is the probability that policy π will select action a in state s. The agent's task is to compute an optimal policy π^* that will allow it to maximize *return*. The expected sum of discounted future reward, or return, for state s while following policy π can be written as:

$$\begin{aligned} V^\pi(s) &= E_\pi\Big\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t = s\Big\} \\ &= \sum_a \pi(s,a)\sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^\pi(s')], \end{aligned}$$

where $\gamma \in [0, 1)$ is the discount rate.

The optimal value function, denoted as V^*, satisfies the Bellman optimality equation:

$$\begin{aligned} V^*(s) &= \max_{\pi} V^{\pi}(s) \\ &= \max_{a} \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^*(s')] \end{aligned}$$

for all $s \in S$.

Action-value or Q-value functions explicitly represent the value of a state-action pair. $Q^{\pi}(s, a)$ is the expected return of starting in state s, taking action a, and following policy π from that point on:

$$\begin{aligned} Q^{\pi}(s, a) &= E_{\pi}\Big\{ \sum_{k=0}^{\infty} \gamma^k r_{i+k+1} | s_t = s, a_t = a \Big\} \\ &= R^a_{ss'} + \gamma \sum_{s'} P^a_{ss'} \sum_{a'} \pi(s, a') Q^{\pi}(s', a'). \end{aligned}$$

The optimal Q-value function can be written as:

$$\begin{aligned} Q^*(s, a) &= E\big\{ r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') | s_t = s, a_t = a \big\} \\ &= \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma \max_{a' \in A(s')} Q^*(s', a')] \end{aligned}$$

for all $s \in S, a \in A$.

The two value functions are related by $V^*(s) = \max_a Q^*(s, a)$, and the use of either type of value function allows an agent to act optimally. However, the V-function requires the agent perform a one-step look-ahead search to select an action. This search is difficult if the agent does not have an accurate transition model. The Q-function is defined over state-action pairs, rather than just states, and requires more storage space than the V-function. However when using the Q-function, the agent does not need the transition model since the value of an action is explicitly represented.

2.2 Reinforcement Learning

Reinforcement Learning (RL) (Sutton & Barto, 1998) is a machine learning framework in which an agent learns to take actions in its environment in order to maximize some measure, such as its long term discounted reward. At each time-step t the agent perceives the state of its environment $s_t \in S$ and selects an action from the set of available actions $a_t \in A$. In response to this action a reward r is given, and the agent transitions to the next state s_{t+1}.

Many algorithms in the RL framework can be viewed as variants of temporal-difference (TD) learning. Using TD methods, the agent learns estimates of a value function directly from experience in the environment without a model of the environment's dynamics. TD methods use a bootstrapping technique to update the estimates; estimates are updated using existing estimates.

Q-learning (Watkins, 1989), a popular TD-type RL algorithm, approximates the optimal action-value function through experience. Suppose the agent observes a current state s, executes action a, receives reward r, and then observes state s'. Q-learning updates the current estimate $Q_t(s, a)$ of $Q^*(s, a)$ using the following update:

$$Q_{t+1}(s, a) \leftarrow Q_t(s, a) + \alpha[r_t + \gamma \max_{a' \in A(s')} Q_t(s', a') - Q_t(s, a)],$$

where $\alpha \geq 0$ is the step-size parameter.

2.3 Value Function Approximation

As the state or state-action space grows larger, it becomes computationally infeasible to fully represent the value function. Function approximation maps a state s and action a into a length k feature vector $\phi(s, a)$ where $k \ll |S \times A|$ and $\phi(s, a) \in \mathbb{R}$.

In this dissertation, we focus on linear function approximation; Q is approximated a weighted linear combination of feature vectors

$$\hat{Q}^{\pi}(s,a|\boldsymbol{\theta}) \quad = \quad \sum_{j=1}^{k} \phi_j(s,a)\theta_j, \tag{2.1}$$

where θ_j is the j-th parameter and $\boldsymbol{\theta}$ is a vector of the k parameters. Φ is a matrix with $|S| \times |A|$ rows and k columns such that $\phi_j(s,a)$ is a row of this matrix. Since Q is linear in $\boldsymbol{\theta}$, there is exactly one optimal $\boldsymbol{\theta}$ (or in degenerate cases, one set of equally good optima). However, because the basis functions that form the columns of Φ may be arbitrarily complex, it is possible to represent any value functions.

Algorithms to learn the value function now update the parameter vector instead of the tabular values. Q-learning can be modified to update $\hat{Q}$ by updating the parameters in the following way:

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \alpha[r_t + \max_{a' \in A(s')} \gamma \hat{Q}_t(s',a'|\boldsymbol{\theta}_t) - \hat{Q}_t(s,a|\boldsymbol{\theta}_t)] \cdot \nabla_{\boldsymbol{\theta}_t} \hat{Q}_t(s,a|\boldsymbol{\theta}_t).$$

This is a gradient descent approach to learn the parameters, where α determines the step size.

2.3.1 Least-squares Methods

Least-squares projection (Bradtke & Barto, 1996; Boyan, 1999; Nedic & Bertsekas, 2003; Lagoudakis & Parr, 2003) is a parameter estimation method used to approximate value functions. Boyan (1999) examined the link between least-squares and TD methods. Both approaches are solving the same system of equations, however TD methods are descending a gradient towards the solution. TD methods are cheaper but are not as sample efficient. Least-squares methods do not require specifying a step-size parameter α.

One such method, Least Squares Policy Iteration (LSPI) (Lagoudakis & Parr, 2003), is a least-squares temporal-difference learning method for approximating the action-value function $\hat{Q}^{\pi}$. LSPI approximates the action-value function by projecting the exact Q-function onto a subspace spanned by a set of basis functions $\phi(s,a)$. LSPI approximates the true state-action value function $Q^{\pi}(s,a)$ using the linear architecture defined in Equation 2.1.

LSPI uses a linear approximation scheme that attempts to find a fixed point approximation of the Bellman equation $T_\pi Q^\pi \approx Q^\pi$, where T_π is the Bellman operator, which is defined as:

$$T_\pi(Q)(s,a) = R_{ss'}^{\pi(s)} + \gamma \sum_{s' \in S} P_{ss'}^{\pi(s)} \sum_{a' \in A(s')} \pi(s',a') Q(s',a')$$

where $\pi(s',a')$ is the probability that policy π selects action a' in state s'.

The fixpoint solution for the parameters is given by:

$$\boldsymbol{\theta}^\pi = (\Phi^T(\Phi - \gamma P \Pi_\pi \Phi))^{-1} \Phi^T R,$$

where γ is the discount factor, and Π_π is a stochastic matrix that describes the current policy π. The parameters $\boldsymbol{\theta}^\pi$ minimize the projected Bellman residual in the subspace spanned by the basis functions. LSPI begins with an initial policy π_0 and an initial set of parameters $\boldsymbol{\theta}_0$. The algorithm repeatedly iterates until the parameter vector converges.

2.3.2 Hand-Coded Basis Functions

Thus far, we have described how the action-value function can be approximated and how the weight vector can be learned from experience. However, we have not described how Φ is constructed. Constructing Φ is critical, as it is the representation the agent uses for learning and defines the similarity between states. In this subsection, we briefly describe traditional techniques where the basis functions are designed a priori.

There has been a great deal of work on hand-coded basis functions in RL, including CMACs (Sutton & Barto, 1998), radial basis functions (Sutton & Barto, 1998; Lagoudakis & Parr, 2003), polynomials (Lagoudakis & Parr, 2003), Fourier basis functions (Konidaris & Osentoski, 2008) and nonlinear approaches, such as neural nets (Tesauro, 1992). Most approaches use a fixed predefined parametric representation and then use a parameter estimation technique such as temporal difference learning (Tsitsiklis & Van Roy, 1997; Sutton

& Barto, 1998), least squares projection (Bradtke & Barto, 1996; Boyan, 1999; Nedic & Bertsekas, 2003; Lagoudakis & Parr, 2003), or linear programming (de Farias & Van Roy, 2003; Guestrin et al., 2003) to approximate the value function.

There has also been work on using non-parametric techniques for value function approximation. Gordon (1995) used nearest neighbor methods for value function approximation, and Ormoneit and Sen (2002) used kernel density estimation. Our approach differs from these by modeling the underlying manifold of the data and extracting a distance metric that respects this manifold. Kernel methods have been applied to value function approximation through the use of support vector machines (Dietterich & Wang, 2002) and Gaussian processes (Engel et al., 2003; Rasmussen & Kuss, 2004). Our approach is similar to these approaches; however, we do not use a hand-engineered kernel and instead use a data-dependent graph or diffusion kernel (Kondor & Vert, 2004).

Some research focuses on dynamically defining the basis functions upon the state space. Singh et al. (1995) introduced the concept of soft state clustering for function approximation. In this approach, the function approximator is a set of clusters over the state space. Soft state clustering allows a state to belong to several clusters. They introduce an Adaptive State Aggregation (ASA) algorithm and define a good clustering as one that reduces the Bellman error for the states of the MDP. They demonstrate that their ASA algorithm is able to construct a clustering that reduces the Bellman error on the state space. Kretchmar and Anderson (1999) investigated a technique to automatically allocate basis functions to regions of the state space based on the probability of visiting the regions. Smith (2002) used Self-Organizing Maps (SOMs) to map MDPs with continuous states and continuous actions to a smaller space of discrete states and discrete actions. Separate SOMs were used for states and actions, and the SOMs were updated based upon the performance of the agent. SOMs were selected in order to capture some of the topology of these spaces; however, this work did not directly attempt to capture the underlying manifold of the state or state-action space.

Driessens et al. (2006) used Gaussian processes in conjunction with graph kernels as a function approximation for reinforcement learning. This approach is a non-parametric Bayesian technique; no prior assumption is made about the parameters. Regression is used to set the parameters in relational reinforcement learning tasks.

Some work has investigated representing structure in value functions themselves. Foster and Dayan (2002) investigated finding common structure in MDPs by decomposing a set of value functions according to shared structure. Their work uses a mixture of Gaussians as the model and trained the model using EM-based maximum likelihood techniques. Drummond (2002) uses computer vision techniques to find nonlinearities in the value function.

Menache et al. (2005) propose modifying both the basis functions and parameter vector simultaneously. They suggest two approaches: a gradient based adaptation and a cross entropy approach and evaluate their approach using RBFs. The gradient method was found to be more susceptible to local minimum and the cross entropy performed significantly better but required a greater computational effort. This approach may not result in a linear function approximator.

2.4 Representation Discovery

Currently, most research in machine learning assumes that the learning algorithm has a set of useful features that were provided by a domain expert. In this dissertation, we are interested in algorithms that allow the agent to automatically discover representations from its experience and use these representations for learning. Representation discovery is an area of vital importance for machine learning and artificial intelligence (Mahadevan, 2008). The proper representation facilitates learning. In this section, we first discuss dimensionality reduction and manifold learning. We then review automatic basis function construction in Markov decision processes.

2.4.1 Dimensionality Reduction and Manifold Learning

Many machine learning applications involve large data sets with a significant amount of data. Often this data has a high dimension, or number of variables. One reason that high dimensional data can be be difficult for machine learning algorithms is because many variables may not be important for the task of interest. The goal of dimensionality reduction is to construct lower dimensional representations that capture the important features of the data. Dimensionality reduction approaches can be categorized as either representing the data on a subset of the original features – feature subset selection – or by constructing new features. Many approaches to basis function construction can be thought of as dimensionality reduction techniques. Rather than learning the full set of parameters required in a table look-up, the size of the space is reduced and then the smaller set of parameters is learned. In this section, we will primarily focus our discussion on approaches to dimensionality reduction that construct new features since they are most closely related to the approach we will use for basis function construction. We only briefly mention a few approaches, more information about different dimensionality reduction techniques is available in several survey papers (Fodor, 2002; Ye, 2003; Gorban et al., 2007).

Traditional dimensionality reduction techniques often focus on finding reductions that are linear transformations of the data. Principal component analysis (PCA) (Jolliffe, 1986) and singular value decomposition (SVD) (Golub & Loan, 1989) find a low dimensional fitting of the data that minimizes the mean square error. These approaches finds principal components, which are the eigenvectors of the covariance matrix. Factor analysis (Mardia et al., 1995) also finds a linear reduction but uses second order information; and projection pursuit (Huber, 1985) captures higher than second order information such as the negative Shannon entropy. Independent component analysis (Hyvvärinen, 1999) finds linear projections that are as statistically independent as possible, which is a stronger condition than the correlation conditions of previously discussed approaches.

In the machine learning community there has been increasing interest in manifold and spectral learning techniques for nonlinear dimensionality reduction. Isomap (Tenenbaum et al., 2000), locally linear embedding (LLE) (Roweis & Saul, 2000), and Laplacian Eigenmaps (Belkin & Niyogi, 2001) are unsupervised nonlinear dimensionality reduction techniques. These techniques assume that local distance metrics are given but global distances are unknown. They learn the underlying structure of the manifold by maintaining local neighborhood structures. These techniques have been found to be especially useful in domains such as vision and text where the data set is assumed to lie on a low dimensional manifold.

The work on nonlinear dimensionality reduction has been applied to semi-supervised learning (Belkin & Niyogi, 2004). In this work, the unlabeled data are used to discover the underlying manifold of the data. The labeled examples are then used to develop a classifier defined over the manifold. The major difference between this work and ours is that reinforcement learning is an active learning process that does not take place on a static data set. Our work is also different in that it is specifically aimed toward solving MDPs and seeks to approximate value functions over a state space graph.

A large amount of the research on dimensionality reduction has mainly focused on classification and clustering; however, some work has focused on low dimensional embedding in situations with dynamics or tasks involving optimal control. We specifically describe some examples of this type of work since it also constructs low dimensional representations from data collected from an agent taking actions in a domain.

Jenkins and Matarić (2004) introduced spatio-temporal Isomap, an extension to Isomap for data with both spatial and temporal relationships. The approach was used to find low dimensional data from a teleoperated humanoid robot and from motion capture data of humans performing different activities. Tsoli and Jenkins (2007) and Ciocarlie et al. (2007) investigate dimensionality reduction techniques for grasping tasks. Ferris et al. (2007); Ham et al. (2005) use dimensionality reduction techniques for robotic localization tasks.

Yairi (2007) compares multiple dimensionality reduction approaches on a robotic map building task without localization.

Shi and Malik (2000) introduce NCuts, an approach that uses spectral techniques to perform image segmentation. Grudic and Mulligan (2005) use dimensionality reduction techniques to perform clustering in visual tasks.

Action Respecting Embedding (ARE) (Bowling et al., 2005) is an approach that uses actions when building low-dimensional representations of data. In this approach, the data are transformed into a low-dimensional representation in which actions are a simple transformation in the new space. While actions are used to create the embedding, they are not explicitly represented in this approach.

2.4.2 Automatic Basis Function Construction in Markov Decision Processes

Recently ideas from the manifold learning and dimensionality reduction literature have been used to build basis functions for Markov decision processes. Much of this work has focused on modeling the intrinsic structure of the domain, particularly the state space. The goal of this work is to automatically construct basis functions such that the solution of the MDP calculated using these basis functions closely approximates the solution of the original MDP.

Some early work recognized that the state space of an MDP might be embedded in a low-dimensional manifold. This early work relied upon heuristics that attempt to exploit this intuition. Smart (2004) proposed the use of manifold techniques for value function approximation. This work used charts to cover the state space, and basis functions were created from the embeddings of the charts. Ratitch and Precup (2004) used Sparse Distributed Memories (SDMs) to create basis functions over the state space. Both of these approaches have similarities to CMACs. However, the location and size of the tiling is dynamic and can be adjusted. While the authors recognized that the MDP could be rep-

resented in a low-dimensional manifold, their approaches are closer to automatic model selection than automatic basis function construction.

Representation Policy Iteration (RPI) (Mahadevan, 2005) constructs a graph over the sampled state space and uses spectral analysis of the graph to define basis functions. Our work specifically builds upon RPI, and we will be discuss this approach in significant detail in Chapter 3. Sugiyama et al. (2007) defines Gaussian kernels on a graph created on the state space and uses them for value function approximation. A technique, similar to the techniques used by Gärtner et al. (2003) and Driessens et al. (2006), is used to extend the basis functions to state-action pairs by incorporating information about the transition probabilities. This approach requires explicitly modeling the transition matrix and performs poorly in highly stochastic environments. An automatic approach for the placement of the Gaussian centers is not currently given. Thus this approach is not a fully automatic basis function construction technique. Several techniques for automating RBF placement have been examined and could be incorporated into this approach (McLoone et al., 1998; Moody & Darken, 1989; Sanchez, 1995; Karayiannis, 1999; Haykin, 1999; Gonzalez et al., 2003; Lazaro et al., 2003).

Other approaches introduced techniques for learning basis functions that incorporates information about the reward function. We call basis functions created by these techniques *reward sensitive basis functions*; basis functions constructed without using the reward function are *reward insensitive basis functions*. If the reward function changes but the state and action spaces and the transition model P remain unchanged, a new set of basis functions must be learned. Keller et al. (2006) and Parr et al. (2007) investigate techniques that learn basis functions that incorporate reward from a specific task by estimating the Bellman residual. Petrik (2007) combines Krylov bases and Laplacian bases to create basis functions that incorporate the reward function. Mahadevan (2009) investigates the use of Drazin bases for value function approximation.

Another vein of research has examined multi-scale basis function construction. These approaches create a hierarchy of basis functions, where the hierarchy contains basis functions at different levels of resolution. One approach employs multigrid methods, typically used to solve differential equations, to construct basis functions at multiple levels of resolution (Ziv, 2004; Ziv & Shimkin, 2005) . Diffusion wavelets (Mahadevan & Maggioni, 2006; Maggioni & Mahadevan, 2006a) are another approach used to automatically construct basis functions over the MDP's state space. This approach compactly represents dyadic powers of the transition matrix at each level of the hierarchy.

2.5 State Abstraction in Reinforcement Learning

In the previous section, we discussed methods to automatically construct basis functions for MDPs. Another approach to compressing the state space in MDPs is state abstraction or state aggregation. Abstraction is frequently described as mapping the original representation of the problem to an *abstract* representation of the problem, where the abstract representation is more compact. This problem has been extensively studied in the context of decision making (Rogers et al., 1991; Giunchiglia & Walsh, 1992). In this section, we briefly review some of the approaches that have been used for abstraction in RL. Li et al. (2006) provide a unified treatment of some of these approaches and a detailed review of different abstraction approaches.

State aggregation methods can be divided into two groups, exact and approximate methods. . Exact methods preserve P and R of the original MDP while constructing the abstract model. Model minimization (Givan et al., 2003) and MDP homomorphisms (Ravindran, 2004; Ravindran & Barto, 2003; Wolfe & Barto, 2006) typically fall into this category of state abstraction.

Approximation methods for homomorphisms have been suggested (Dean et al., 1997; Givan et al., 2000; Ferns et al., 2004). These methods define a similarity measure, and aggregation is performed according to the measure. Adaptive aggregation (Castañon &

Bertsekas, 1989) groups states with similar Bellman residuals. Some work has examined creating abstractions when the agent does not know P and R. This work has mainly involved using statistical tests. The G algorithm (Chapman & Kaelbling, 1991) aggregates states with the same reward and Q-values for each action. The U-tree algorithm (McCallum, 1995) combines states that have the same optimal actions and similar Q-values for the actions. Policy Irrelevance (Jong & Stone, 2005) group states that have the same optimal action. However this approach may not be as useful for learning the optimal policy in the original MDP.

Approaches to abstraction and function approximation attempt to reduce the size of the learning problem by coming up with a more compact representation. The basis matrix in state aggregation approaches partitions the space, where each state can only be in one partition. Most state abstraction approaches require the agent to have access to P and R; however, some work has been done on lifting this assumption. Traditional function approximation techniques have not focused on preserving properties of P or R but also do not require that these functions be known or estimated. However, this work has primarily focused on using the abstractions created exclusively for value function approximation. The use of the basis functions created by these approaches for other applications is not well explored.

CHAPTER 3

SPECTRAL BASES ON GRAPHS

This dissertation focuses on representation discovery in MDPs. We build upon work in representation discovery using the graph Laplacian for constructing basis functions, specifically Representation Policy Iteration (RPI) (Mahadevan, 2005). In this chapter, we define the terminology that will be used throughout the remainder of the dissertation and review this approach in detail.

In this chapter, we specifically review the model-free version of RPI, in which the agent does not have access to the transition matrix P or the reward function R. RPI can be seen as a generic algorithm where the agent explores its environment and automatically constructs basis functions. We specifically discuss a version of RPI where the the agent automatically constructs basis functions over the agent's state space.

Definition 3.1 Automatic State Space Basis Construction Problem: *Given a Markov Decision Process $M = (S, A, P, R)$, automatically construct a low-dimensional representation Φ such that the size of Φ is $|S| \times k$ where $k \ll |S|$.*[1] *Φ should be constructed such that the solution of M calculated using Φ closely approximates the solution of the original MDP M. Φ can be seen as compressing the state space of the MDP.*

In the RPI approach an agent explores its environment and creates a representation of the sampled state space in the form of a graph. The agent then uses spectral graph

[1] It is important to note that it is possible and desirable for the basis to be defined over a set of samples $\tilde{S} \subseteq S$. When the basis is defined over a subset of the state space, an out of sample extension technique is required for states not in the sample set (Mahadevan et al., 2006).

theoretic approaches to create basis functions. Spectral graph theory provides an analytical approach to deducing the principal properties and structure of a graph from its eigenvalues and eigenvectors. In this chapter, we provide a brief overview of spectral graph theory; a more in-depth explanation can be found in Chung (1997). We also give a brief overview of the RPI framework.

3.1 Basic Definitions

We start by first defining a graph and terms commonly associated with a graph. A weighted undirected graph is defined as a tuple $G_u = (V, E, W)$, where V is the vertex set, E is the edge set, and $W : E \rightarrow \mathbb{R}$ is the weight function, where $W(u, v) = W(v, u)$ and $W(u, v) \geq 0$. If $W(u, v) = 0$, there is no edge between vertex u and vertex v. Figure 3.1 shows an example of an undirected graph with five vertices and six edges.

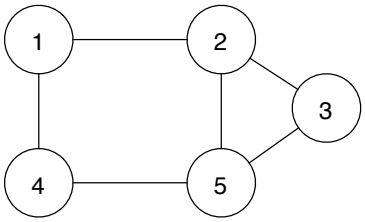


Figure 3.1. An example of an undirected graph containing five vertices and six edges. All edges have a weight of one.

The adjacency matrix for our example graph is :

$$W = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}.$$

The *degree* of vertex u is $d_u = \sum_v W(u,v)$. In our example graph, $d_2 = 3$ and $d_3 = 2$. The *valency matrix* D is a diagonal matrix whose entries are the row sums of W, or equivalently the degree of the vertices. D for our example graph is:

$$D = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 3 \end{bmatrix}.$$

A *walk* of length l on G_u is a sequence of vertices $(v_0, v_1, ...v_l)$ such that $\{v_i, v_{i+1}\} \in E$ for i, $1 \leq i < l$. A *random walk* is a walk on G where v_{i+1} is chosen uniformly at random from the neighbors of v_i. The random walk is defined by a transition probability matrix $\mathcal{P} = D^{-1}W$. $\mathcal{P}(u,v) = \frac{W(u,v)}{d_u}, d_u \neq 0$ denotes the probability of moving from vertex u to vertex v. $\mathcal{P}(u,v) = 0$ if no edge exists between u and v, and $\sum_u \mathcal{P}(u,v) = 1$. $\mathcal{P}$ for our example graph is :

$$\mathcal{P} = \begin{bmatrix} 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ \frac{1}{3} & 0 & \frac{1}{3} & 0 & \frac{1}{3} \\ 0 & \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 0 \end{bmatrix}.$$

$\mathcal{P}$ is called a diffusion model because for any function f on G_u, the powers of $\mathcal{P}^t f$, where t is the number of steps, determine how quickly the random walk will mix and converge to the stationary distribution $\rho(v) = \frac{d_v}{vol(G)}$, where $vol(G) = \sum_{v \in G} d_v$ is the *volume* (Chung, 1997). However, $\mathcal{P}$ is not necessarily a symmetric matrix, and it is often beneficial, for computational reasons, to find a symmetric matrix that has a closely related spectral structure. This matrix is the graph Laplacian matrix.

3.1.1 Functions over Graphs

Before introducing the graph Laplacian we discuss functions over a graph. A function $f : V \rightarrow \mathbb{R}$ over a graph maps vertices of the graph to the real numbers. We define an inner product between two functions f and g to be:

$$< f, g >= \sum_{v \in V} f(v)g(v).$$

The $\mathbb{L}^2$-norm of a function over G is:

$$||f||_2^2 = \sum_{v \in V} |f(v)|^2 d_v,$$

where d_v is the degree of vertex v. We can now discuss the idea of smooth functions over a graph. Intuitively, a smooth function over a graph means that $f(u)$ will be similar to $f(v)$ if u and v are connected in the graph. We can describe how a function over the graph changes through its gradient. The gradient of a function over a graph is defined as:

$$\nabla_f(u, v) = W(u, v)(f(u) - f(v)).$$

The smoothness of a function over a graph is measured by the Sobolev norm (Mahadevan & Maggioni, 2006):

$$||f||_{\mathcal{H}^2}^2 \quad = \quad ||f||_2^2 + ||\nabla_f||_2^2 = \sum_{v \in V} |f(v)|^2 d_v + \sum_{u \sim v} |f(u) - f(v)|^2 W(u, v). \quad (3.1)$$

$\sum_{u \sim v}$ denotes the sum over all unordered pairs u and v, where u and v are adjacent. This can also be seen as taking the sum over all the edges. It is important to note that each edge is only counted once in this sum.

The first term of the Sobolev norm controls the size, in terms of the $\mathbb{L}^2$-norm, of the function f. The second term, also known as the Dirichlet sum, controls the size of the gradient. The smoother a function f is over the graph the smaller $||f||_{\mathcal{H}^2}^2$ will be.

3.1.2 Graph Laplacian

Intuitively the graph Laplacian measures how information flows throughout the graph. There are two forms of the graph Laplacian (Chung, 1997): the combinatorial Laplacian of a graph is defined as:

$$L = D - W, \tag{3.2}$$

and the normalized Laplacian is:

$$\mathcal{L} = D^{-\frac{1}{2}}(D - W)D^{-\frac{1}{2}}. \tag{3.3}$$

It is often more intuitive to think about the value of these functions for two vertices in the graph. The combinatorial Laplacian can be written as:

$$L(u,v) = \begin{cases} d_u & \text{if } u = v, \\ -1 & \text{if } u \text{ and } v \text{ are adjacent,} \\ 0 & \text{otherwise,} \end{cases}$$

and the normalized Laplacian can be written as:

$$\mathcal{L}(u,v) = \begin{cases} 1 & \text{if } u = v, \text{ and } d_u \neq 0 \\ -\frac{1}{\sqrt{d_u d_v}} & \text{if } u \text{ and } v \text{ are adjacent,} \\ 0 & \text{otherwise.} \end{cases}$$

The normalized Laplacian enforces a normalization constraint on the Laplacian, where the the degree of a vertex is a local measure. The combinatorial Laplacian for our example graph is:

$$L = \begin{bmatrix} 2 & -1 & 0 & -1 & 0 \\ -1 & 3 & -1 & 0 & -1 \\ 0 & -1 & 2 & 0 & -1 \\ -1 & 0 & 0 & 2 & -1 \\ 0 & -1 & -1 & -1 & 3 \end{bmatrix}.$$

An *operator* over a graph takes a function f over the graph and transforms it into another function f' over the graph. The Laplacian can be viewed as an operator on the space of functions $f : fV \to \mathbb{R}$. In particular, the Laplacian can be viewed as a difference operator. When the Laplacian is applied to a function f over the graph, it can be shown that:

$$Lf(u) = \sum_{u \sim v} (f(u) - f(v))W(u, v). \tag{3.4}$$

It can be shown that:

$$\langle f, Lf \rangle = \sum_{u} f(u)Lf(u) = \sum_{u \sim v} (f(u) - f(v))^2 W(u, v). \tag{3.5}$$

This property is important because it means that smoothness is measured based on the connectivity in the graph and not Euclidean space. Additionally, it is worth noting that $\langle f, Lf \rangle$ equals $||\nabla_f||_2^2$, from Equation 3.1.

Earlier we discussed how the transition probability matrix $\mathcal{P}$ is useful for analyzing properties of the graph. At first glance, the graph Laplacian and $\mathcal{P}$ seem to have little in common. However, the two are related. The connections are best understood by examining the normalized Laplacian $\mathcal{L}$:

$$
\begin{aligned}
\mathcal{L} &= D^{-\frac{1}{2}}(D-W)D^{-\frac{1}{2}} \\
&= I - D^{-\frac{1}{2}}WD^{-\frac{1}{2}} \\
I - \mathcal{L} &= D^{-\frac{1}{2}}WD^{-\frac{1}{2}} \\
D^{-\frac{1}{2}}(I-\mathcal{L})D^{-\frac{1}{2}} &= D^{-1}W \\
D^{-\frac{1}{2}}(I-\mathcal{L})D^{-\frac{1}{2}} &= \mathcal{P}
\end{aligned}
$$

From this we can see that the random walk operator and $I - \mathcal{L}$ are similar. In fact, the eigenvectors of the random walk are the eigenvectors of $I - \mathcal{L}$ multiplied by $D^{-\frac{1}{2}}$. We can now provide a rationale for using the eigenvectors of the graph Laplacian as a basis representation for the graph.

3.1.2.1 Spectral Decomposition of the Graph Laplacian

We first give a brief review of eigenvectors and eigenvalues. Given an n by n matrix A, a non-zero vector ϕ is defined to be an *eigenvector* of A if it satisfies

$$A\phi = \lambda\phi, \tag{3.6}$$

where λ is the eigenvalue associated with eigenvector ϕ. If A is symmetric, the eigenvectors of A are linearly independent and form an orthogonal basis of A. A *basis* for a space is defined as a set of vectors such that a linear combination of these vectors can represent every vector in that space, and none of these vectors can be represented as a linear combination of the other eigenvectors. If A has n linearly independent eigenvectors we can use them to diagonalize A:

$$A\Phi = \Phi\Lambda \tag{3.7}$$

$$A = \Phi\Lambda\Phi^{-1}, \tag{3.8}$$

where Λ is a diagonal matrix, $\Lambda(i,i) = \lambda_i$, and Φ is the eigenvector matrix where each column is a distinct eigenvector.

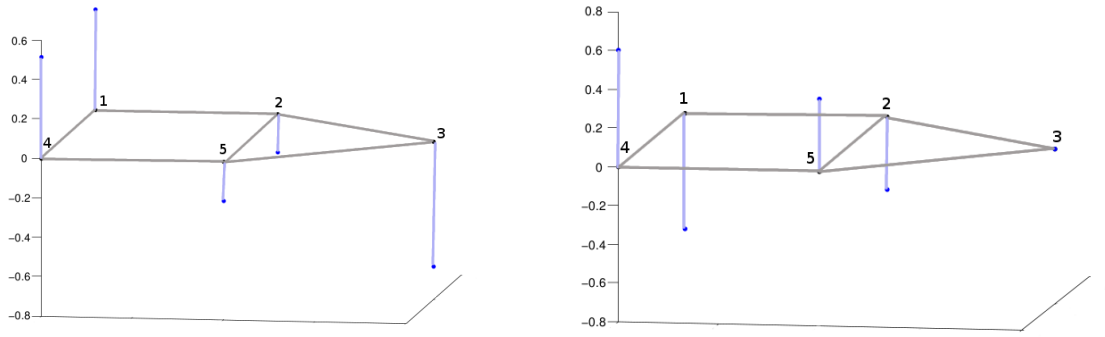
The Laplacian is symmetric as well as positive semi-definite, and thus it has eigenvalues that are real valued and non-negative (Chung, 1997). The first eigenvalue $\lambda_1 = 0$. The first eigenvector of L is a constant function $\phi_1 = c\mathbf{1}$ where c is a constant and $\mathbf{1}$ is a vector of ones. The first eigenvector of $\mathcal{L}$ is $\phi_1 = \sqrt{D}\mathbf{1}$. The eigenvectors of the graph Laplacian are an orthonormal basis that span the wholwholee space of functions. These basis functions are defined over the entire graph and thus capture global features of the graph.

Λ and Φ for L of our example graph are:

$$\Lambda = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1.3820 & 0 & 0 & 0 \\ 0 & 0 & 2.3820 & 0 & 0 \\ 0 & 0 & 0 & 3.6180 & 0 \\ 0 & 0 & 0 & 0 & 4.6180 \end{bmatrix}$$

$$\Phi = \begin{bmatrix} -0.4472 & 0.5117 & -0.6015 & -0.1954 & -0.3717 \\ -0.4472 & -0.1954 & -0.3717 & 0.5117 & 0.6015 \\ -0.4472 & -0.6325 & -0.0000 & -0.6325 & 0.0000 \\ -0.4472 & 0.5117 & 0.6015 & -0.1954 & 0.3717 \\ -0.4472 & -0.1954 & 0.3717 & 0.5117 & -0.6015 \end{bmatrix}.$$

Figure 3.2 shows a visualization of the 2nd and 3rd eigenvectors in Φ. In this visualization, the vertices of the graph are labeled and the grey lines show the graph's edges. The dark blue points are the values of the eigenvector for each vertex in the graph. To enable easier visualization, we added light blue lines from each vertex in the graph to its value in the eigenvector. If no line appears, the value of that vertex in the eigenvector is zero.



(a) 2nd Eigenvector

(b) 3rd Eigenvector

Figure 3.2. A three dimensional view of the 2nd and 3rd eigenvectors of the graph Laplacian for the graph in Figure 3.1. Each point represents the value of the eigenvector for that vertex. We added lines from the graph to the points to help with visualization.

3.1.3 Embeddings of the Graph

The *embedding* of a vertex, $\Phi(v)$, is the spectrum evaluated at vertex v. The embedding of a vertex is its values in any subset of the eigenvectors. However, we will typically refer to the eigenvectors that correspond to the first k low valued eigenvalues. The embedding of vertex 3 in our graph when $k = 3$ is

$$\Phi(3) = [-0.4472 \quad -0.6325 \quad -0.0000].$$

The embedding of the vertices can be used to visualize the graph in a two dimensional plane. When we refer to the embedding of a graph, we refer to the visualization of the graph such that the vertices of the graph are plotted according to their values in the 2nd and 3rd eigenvectors. Figure 3.3 shows the embedding for our example.

3.1.4 Directed Graph Laplacian

Thus far, our discussion has been limited to undirected graphs. However, there are many scenarios where a directed graph would be more appropriate. In this section, we briefly summarize spectral decomposition of the Laplacian on directed graphs; a more in depth analysis can be found in (Chung, 2005; Johns & Mahadevan, 2007).

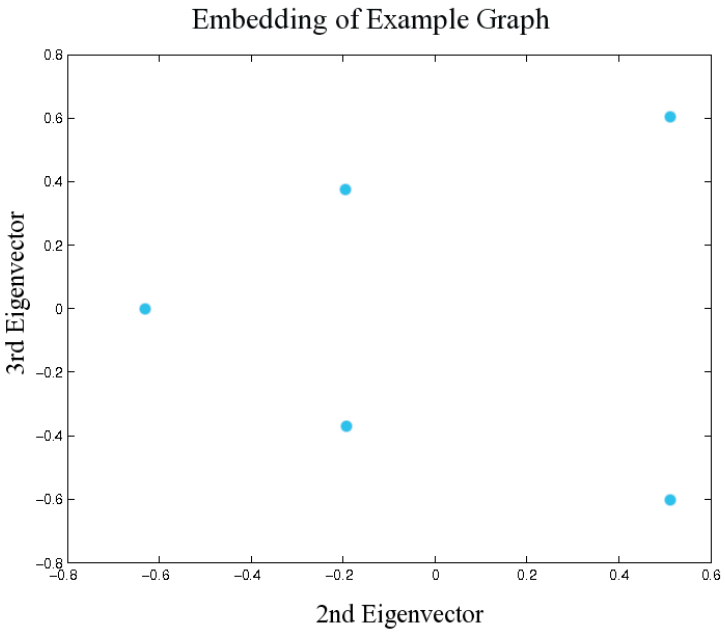


Figure 3.3. The embedding of the graph in Figure 3.1.

A weighted directed graph is defined as a tuple $G_d = (V, E_d, W)$ where V is the set of vertices, E_d is the set of directed edges, and W is the weight matrix. The major distinction between the directed and undirected graph is the non-reversibility of the edges. A directed graph may have weights $W(u, v) = 0$ and $W(v, u) \neq 0$. Figure 3.4 shows an example of a directed graph. This example is similar to that shown in Figure 3.1. However, in the new graph four of the edges are directed: $E(2, 3)$, $E(2, 5)$, $E(3, 5)$, and $E(4, 1)$.

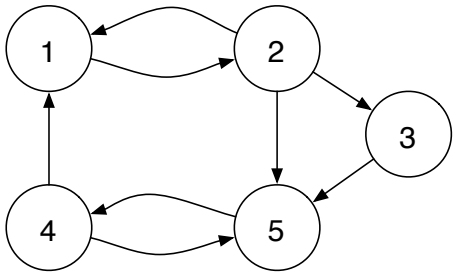


Figure 3.4. An example of an directed graph containing five nodes, two undirected edges, and four directed edges. All edges have a weight of one.

The weight matrix and valency matrix for the graph in Figure 3.4 are:

$$W = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad \text{and} \quad D = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

To define the graph Laplacians on G_d, we must first introduce the Perron vector, ψ, which is used to make the transition matrix symmetric. The transition probability matrix of G_d is defined as $\mathcal{P} = D^{-1}W$. The probability transition matrix for the graph in Figure 3.4 is:

$$\mathcal{P} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ \frac{1}{3} & 0 & \frac{1}{3} & 0 & \frac{1}{3} \\ 0 & 0 & 0 & 0 & 1 \\ \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

The Perron-Frobenius Theorem states that if G_d is strongly connected, then $\mathcal{P}$ has a unique left eigenvector ψ with all positive entries such that $\mathcal{P}\psi = \rho\psi$, where ρ is the spectral radius. The spectral radius of a matrix A is the real number $\max\{|\lambda| : Ax = \lambda x\}$, where $|\lambda|$ is the modulus or formal product of the (possibly complex-valued) eigenvalue λ (Mahadevan, 2009). ρ can be set to 1 by normalizing ψ such that $\sum_i \psi_i = 1$. A more intuitive way of thinking of ψ is as the stationary distribution of a random walk on the graph. The example graph is strongly connected since there is a path from all vertices to all other vertices within the graph.

There is no closed-form solution for ψ; however, there are several algorithms to calculate it. The power method (Golub & Loan, 1989) is an approach to iteratively calculate ψ that starts with an initial guess for ψ, uses the definition $\psi\mathcal{P} = \psi$ to determine a new estimate, and iterates. Another technique is the Grassman-Taksar-Heyman (GTH) algorithm.

This technique uses a Gaussian elimination procedure designed to be numerically stable. The naive GTH implementation runs in $O(n^3)$, but this can be improved in $O(nm^2)$ if $\mathcal{P}$ is sparse. Other techniques, such as Perron complementation (Meyer, 1989), have been introduced to speed up convergence.

Ψ is a diagonal matrix where $\Psi_{ii} = \psi_i$. For our example Ψ is

$$\Psi = \begin{bmatrix} .2 & 0 & 0 & 0 & 0 \\ 0 & .2 & 0 & 0 & 0 \\ 0 & 0 & .0667 & 0 & 0 \\ 0 & 0 & 0 & .2667 & 0 \\ 0 & 0 & 0 & 0 & .2667 \end{bmatrix}.$$

The graph Laplacians for the directed graph are defined by Chung (2005) as

$$L_d = \Psi - \frac{\Psi\mathcal{P} + \mathcal{P}^T\Psi}{2} \tag{3.9}$$

and

$$\mathcal{L}_d = I - \frac{\Psi^{1/2}\mathcal{P}\Psi^{-1/2} + \Psi^{-1/2}\mathcal{P}^T\Psi^{1/2}}{2}. \tag{3.10}$$

L_d for the graph in Figure 3.4 is:

$$L_d = \begin{bmatrix} .2 & -.1333 & 0 & -.0667 & 0 \\ -.1333 & .2 & -.0333 & 0 & -.0333 \\ 0 & -.0333 & .0667 & 0 & -.0333 \\ -.0667 & 0 & 0 & .2667 & -.2 \\ 0 & -.0333 & -.0333 & -.2 & .2667 \end{bmatrix}.$$

L_d is a symmetric matrix, and we can work backwards from L_d to find the analogous symmetric graph. We show this graph in Figure 3.5; note that it now has undirected edges

with weights. These weights constrain the random walk. Recall that the random walk is defined as $\mathcal{P}(u, v) = \frac{W(u,v)}{d_u}$. Vertex u has a lower probability of moving to a vertex v when an edge with a low weight connects them. This example illustrates how the directed graph Laplacian essentially can be seen as making the directed graph undirected such that the properties of the random walk are preserved.

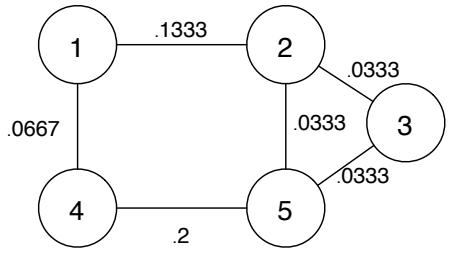


Figure 3.5. The analogous symmetric graph. The edge weights, listed on the edges, help maintain the properties of the random walk of the original directed graph.

The directed Laplacian requires a strongly connected graph. However, graphs created from data may not have this property. In order to ensure that this property exists, we use a teleporting random walk (Page et al., 1998). With probability η the agent acts according to the transition matrix $\mathcal{P}$, and with probability $1 - \eta$ teleports to any other vertex in the graph uniformly at random. This assumption is not built into the domain or the data. It is only used for the purpose of creating ψ and performing the spectral decomposition.

3.2 Applications of Spectral Graph Analysis

Thus far, we have discussed graphs and spectral graph analysis as an abstract concept. Graphs are a natural representation for the data of interest for machine learning applications. Graphs can be used to model interactions between people, the relationship between documents, and the relationship between portions of images or images themselves. Once a graphical representation of the data has been constructed, the techniques and analysis we discussed earlier in this section can be performed. Spectral analysis of graphs can be used

in many ways. Some popular applications of spectral theory are graph partitioning, graph compression, and function approximation.

Spectral graph partitioning is performed by using the eigenvector that corresponds to the 2nd smallest eigenvalue of the graph Laplacian. This eigenvector is often referred to as the Fiedler vector. The Fiedler vector can be used to partition the vertices into two sets and used recursively to cluster the vertices of the graph. Spectral graph partitioning has been used in many applications such as vision (Shi & Malik, 2000) , text (Dhillon, 2001), clustering (Ng et al., 2002), and robotics (Olson et al., 2005).

Graph compression and function approximation can be seen as essentially analogous approaches. Graph compression uses the eigenvectors that correspond to the smallest k eigenvalues. The graph with n vertices goes from being represented in an n by n matrix to an n by k matrix where $n \gg k$. Function approximation compactly represents a function defined over the entire graph by representing the values over the compressed graph. These approaches have been used in may applications such as graphics (Karni & Gotsmann, 2000) and semi-supervised learning (Belkin et al., 2004).

3.3 Representation Policy Iteration

The major insight of representation policy iteration (RPI) (Mahadevan, 2005; Mahadevan & Maggioni, 2007) is that the agent can use its experience in the domain to construct compact representations of the MDP. The initial work used graphs to compress MDPs as well as to approximate any function over the state space of the MDP. This work was extended to use diffusion wavelets (Maggioni & Mahadevan, 2006b) and Drazin bases (Mahadevan, 2009).

In this section, we specifically review the approach to the construction of basis functions where basis functions are constructed from the graphs and used as a representation of the state space of the domain. The eigenvectors of the graph Laplacian can be used for value function approximation in MDPs. Previously, RL algorithms primarily used hand en-

gineered basis functions such as RBFs and polynomial basis functions. RPI automatically constructs basis functions from the agent's experience in the domain. Basis functions are constructed via spectral analysis of the state graph built from the MDP. The RPI algorithm is illustrated in Figure 3.6. The first phase is an initial sample collection according to some initial policy; the second is the basis construction phase, and the third is the control learning phase. Figure 3.7 shows a more detailed algorithmic view of the RPI algorithm.

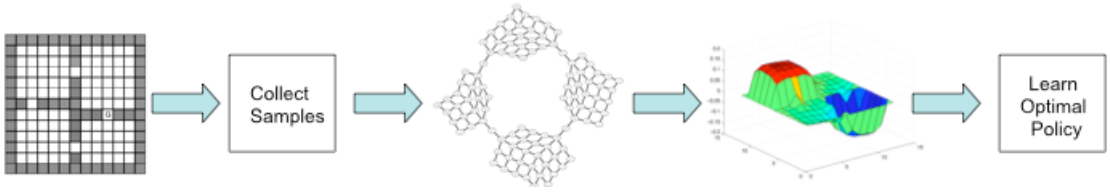


Figure 3.6. An illustration of the general approach in which the agent starts in a domain, collects samples via exploration, builds a graph, calculates the k smallest eigenvectors of the graph Laplacian and uses the eigenvectors as basis functions to represent the value function during learning.

3.3.1 MDPs as Graphs

RPI builds a graph of an MDP, as defined in Section 2.1. First, we define a graph over an MDP.

Definition 3.2 MDP State Graph: *An MDP state graph G is defined over an MDP M, such that the vertices V correspond to the states S or a subset of the states. An edge exists between u and v if there is an action that causes a transition between the corresponding states. The weights on the edges of the graph can be defined in many ways. For simplicity, unless otherwise stated, we assume that that $W(u, v) = 1$ if u and v are connected, otherwise $W(u, v) = 0$.*

An MDP state graph can easily be constructed from the transition model P. However, in RL an agent does not typically have access to P. RPI assumes that the agent performs an initial exploratory period. During this time, the agent executes a policy π_m, typically

a random walk, and collects a set of samples $\mathcal{D}$, each of which consists of state, action, reward and next state, (s, a, r, s'). A graph is built such that the vertices of the graph correspond to the states in $\mathcal{D}$. States i and j are connected if they are linked temporally in $\mathcal{D}$: $W(i, j) = 1$ if i and j are linked otherwise $W(i, j) = 0$.

It is important to note that in RPI, the agent is not building an accurate estimate of P, but is building an approximate representation of state transitions. This places the approach somewhere between the realm of model free and model based approaches. Building an accurate estimate of P would require significantly more samples than is required with this approach.

The premise behind this approach is that building representations initially makes it easier for learning to occur at a later time. RPI assumes that while the reward function is typically not smooth, the value function will often be smooth over the state space. The graph models the geometry of the state space, and thus the spectral approaches we described for general graphs can be applied to the MDP state graph. Phi the basis functions are created using the first k eigenvectors of the graph Laplacian. The eigenvectors may be used to approximate the value function as described in Section 2.3.

This approach has been demonstrated to significantly improve performance over traditional basis function approaches such as RBFs and polynomial basis functions (Mahadevan, 2005; Mahadevan & Maggioni, 2007; Mahadevan, 2008). Primarily, this is due to the fact that the basis functions respect the geometry of the state space.

Most previous work on automatic basis function construction has employed least squares learning approaches, particularly LSPI (Lagoudakis & Parr, 2003). The choice in function approximator is independent of the choice in learning algorithm. In this dissertation, we use incremental TD learning algorithms; while least squares approaches have been shown to be sample efficient, incremental algorithms are also desirable.

`Model-Free RPI Algorithm` $(\mathcal{D}, \gamma, \epsilon, k, \pi_m)$:
// $\mathcal{D}$: Source of samples (s, a, r, s')
// γ: Discount factor
// ϵ : Stopping criteria
// k: Number of basis functions
//π_m: Initial policy specified as a weight vector w_0

1. **Sample Collection:**
   (a) Generate a set of samples, $\mathcal{D}$, which consists of a state, action, reward, and next state, (s, a, r, s'). The samples are created using a series of exploratory trajectories using π_m. Typically π_m is a random walk that terminates when an absorbing state is reached or some preset maximum number of steps is reached.
   (b) Subsample $\mathcal{D}$ in order to gain a smaller set of transitions $\mathcal{D}_s$ by some method, random or greedy are typical examples.
2. **Representation Learning:**
   (a) Build an undirected weighted graph G_u from $\mathcal{D}_s$ where V is the set of vertices, E_u is the edge set, and W is the weight matrix. The vertices are the set of states, $S \in \mathcal{D}_s$. Several methods can be used to connect the states. The simplest technique is placing an edge with weight 1 between state i and state j if they are temporally linked in $\mathcal{D}_s$.
   (b) Calculate the k lowest order eigenfunctions of the (combinatorial or normalized) graph Laplacian operator on G_u. These k eigenvectors are used as the basis functions ϕ.
      i. Form the directed Laplacian per Equation 3.2 or 3.3.
      ii. Calculate ϕ by computing the eigenvectors of the graph Laplacian.
      Create the basis functions for state action pairs by concatenating the state encoding $|A|$ times.
3. **Control Learning Phase**:
   Use a parameter estimation method such as LSPI (Lagoudakis & Parr, 2003) or Q-learning (Watkins, 1989) to find the best policy π. Previous papers have primarily focused on the use of LSPI.
4. **Optional:** Repeat the above procedure by calling `RPI`$(\mathcal{D}, \gamma, \epsilon, k, \pi_{m+1})$

Figure 3.7. The generic model-free RPI algorithm for learning representation and control (Mahadevan & Maggioni, 2007).

CHAPTER 4

REPRESENTATION DISCOVERY USING STATE-ACTION GRAPHS

In this chapter, we discuss algorithms that automatically construct representations for action-value functions. Action-value functions represent the value of taking an action in a given state and can be used to derive an appropriate policy in an MDP, typically by greedily selecting actions with the highest value. Action-value functions are necessary when an agent cannot perform one-step look-ahead search or when this computation is expensive. Action-value functions have a long history in AI and are an important part of reinforcement learning. The most popular type of action value function for RL, the Q-function, was introduced by Watkins (1989). However, the idea of the action-value function predates this work significantly. Shannon (1950) used a function $h(P, M)$ for a chess program to decide if performing move M in position P was worthwhile. In classical physics, Hamilton's principal function is an action-value function (Goldstein et al., 2002).

Watkins (1989) argues for the use of action-value functions because they are significantly smaller to store than models of the reward and transition probability functions. Action-value functions require at most size $|S||A|$. In the worst case, a model of the reward function requires size $|S||A|$, and the transition probability function requires $|S|^2|A|$. However, these functions are often quite sparse and thus compressible. While action-value functions are comparatively smaller, they can still grow to be quite large. Function approximation is necessary when the action-value function cannot be exactly represented or when generalization is desired. Function approximation of an action-value function requires that the basis functions be defined over state-action pairs rather than over states alone.

Most approaches to function approximation create basis functions on the state space and then map these basis functions to state-action space. Perhaps the simplest approach is to copy the state space basis functions for all possible actions, even those not available in the state (Lagoudakis & Parr, 2003; Mahadevan, 2005). Another approach is to use a linear combination of the features of the potential next states of an action as the features for the state-action pair (Gärtner et al., 2003; Driessens et al., 2006; Sugiyama et al., 2008). This approach has been observed to perform well in situations where the environment is deterministic.

Both approaches map features created on the state space to features in state-action space, but the mappings do not allow compression across states and actions simultaneously. In this chapter, we examine how basis functions can be automatically constructed in state-action space.

Definition 4.1 Automatic State-Action Space Basis Construction Problem: *Given a MDP $M = (S, A, P, R)$, automatically construct a low-dimensional matrix representation Φ such that the size of Φ is $|S||A| \times k$ where $k \ll |S||A|$.*[1] *Φ should be constructed such that the solution of M calculated using Φ closely approximates the solution of the original MDP M. Φ can be seen as compressing the state-action space of the MDP.*

We introduce an approach to automatically building basis functions that captures similarities across both states and actions using state-action graphs. We describe these graphs, introduce two techniques for building them, and empirically demonstrate that they perform better than basis functions created from state graphs.

[1] It is important to note that it is possible and desirable for the basis to be defined over a set of samples $\tilde{S} \subseteq S$ and $\tilde{A} \subseteq A$.

4.1 State-Action Space

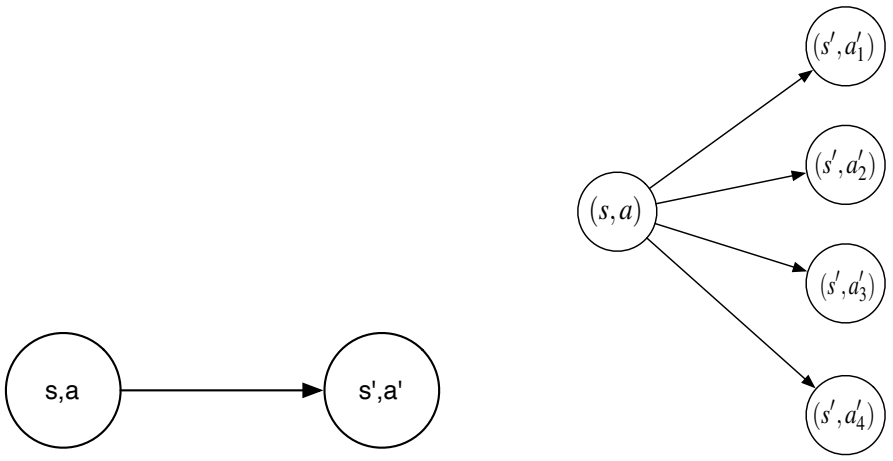
To build basis functions appropriate for action-value functions, it is useful to directly incorporate actions when creating a representation. Most previous approaches construct representations for all actions using copies of the same set of features, with separate weights for each action. In this work, we propose building basis functions using state-action graphs. In our approach the agent can generalize over states and actions simultaneously. This reflects the actual similarities and differences between actions.

Definition 4.2 MDP State-Action Graph: *An MDP state-action graph* $G_{sa} = (V_{sa}, E_{sa}, W_{sa})$ *is defined over an MDP* M *such that each vertex* $v \in V_{sa}$ *corresponds to a state-action pair* (s, a)*, where* $s \in S$ *and* $a \in A$*. An edge exists between* u*,* (s, a)*, and* v*,* (s', a')*, if an action* $a \in A(s)$ *causes a transition between* s *and* s' *and* $a' \in A(s')$*.* W_{sa} *is the weight matrix for the graph and specifies the weights over the edges in* E_{sa}*.*

4.2 Graph Creation in State-Action Space

State-action graphs can be created in a variety of ways. In this section, we assume the agent undertakes an initial exploratory period during which it will collect a set of samples $\mathcal{D}$. These samples will be used to build the state-action graph, G_{sa}. V_{sa} is the set of state-action pairs observed in $\mathcal{D}$.

Two techniques, shown in Figure 4.1, may be used to create E_{sa} in state-action graphs. The first technique, on-policy graph creation, places an edge between (s, a) and (s', a') if $\mathcal{D}$ contains at least one sample where the agent was in state s, took action a, transitioned to state s' and then selected action a'. The second technique, off-policy graph creation, places an edge between (s, a) and (s', a') if $\mathcal{D}$ contains at least one sample where the agent was in state s, took action a, and transitioned to s' where $a' \in A(s')$ is one of the actions available in s'. Self-loops are explicitly excluded in both types of graph creation. On-policy graph creation can be used to model the agent's current policy while the off-policy graph will model the underlying MDP. When the agent is executing a random walk, the two



(a) On-policy graph creation only adds a single edge based on the exact experience of the agent.

(b) Off-policy graph creation adds edges for all actions available in state s'.

Figure 4.1. Two techniques to create state-action graphs.

techniques will converge in the limit to the same graph; however, the off-policy method requires fewer samples.

State-action graphs may be significantly impacted when the environment is stochastic. One approach to handle this is for W_{sa} to be an approximation of P. W_{sa} can be calculated using a simple maximum-likelihood estimation (MLE) approach; we keep track of the frequency of transitions during the exploration period and then divide the edge weight by this number. The MLE approach constructs W_{sa} such that unlikely transitions will not have similar weights as likely transitions. However, this approach does not require an accurate model of P, just an approximate weighting. Figure 4.2 shows simple pseudo-code for the MLE construction of W_{sa}.

4.3 Basis Function Construction Using State-Action Graphs

Once the graph has been created, we use spectral techniques to create basis functions for the state-action graph. State-action graphs are inherently directional. An undirected edge in a state-action graph implies that the agent must be able to transition from state s

State-Action Graph Creation ($\mathcal{D}$):
// $\mathcal{D}$: Source of samples (s, a, r, s', a')
// Creates a graph $G_{sa} = (V_{sa}, E_{sa}, W_{sa})$ where V_{sa} is the set of state-action pairs in $\mathcal{D}$.

// u and (s, a): refer to a state-action pair (s, a) found in $\mathcal{D}$
// v and (s', a'): refers to a state-action pair (s', a') found in $\mathcal{D}$

//$\Upsilon_{\mathcal{D}}(u)$: refers to a function that returns the number of times u is observed in $\mathcal{D}$
//$\Upsilon_{\mathcal{D}}(u, v)$: refers to a function that returns the number of times a transition between u and v occurs in $\mathcal{D}$

If using on-policy graph creation
  $W_{sa}\big((s, a), (s', a')\big) = \Upsilon_{\mathcal{D}}\big((s, a), (s', a')\big)$
Else for off-policy graph creation
  $W_{sa}\big((s, a), (s', a')\big) = 0$
  For all $(s, a, s') \in \mathcal{D}$
    For all $a'' \in A(s')$
      $W_{sa}\big((s, a), (s', a'')\big) = W_{sa}\big((s, a), (s', a'')\big) + 1$
For all $W_{sa}(u, v) \neq 0$
  $W_{sa}(u, v) = W_{sa}(u, v)/\Upsilon_{\mathcal{D}}(u)$

Figure 4.2. Pseudo-Code for creating state-action graphs.

using action a to state s' and transition from state s' with action a' to state s. Assuming an undirected graph as a model leads to a significant number of erroneous edges. Therefore, we use the directed graph Laplacian, described in Section 3.1.4, where the transition matrix is symmetrized using the Perron vector.

We will use the eigenvectors of the directed graph Laplacian on the state-action graph as basis functions during learning. The resulting eigenvectors are directly defined over state-action pairs. The embeddings of the state-action graph are in a different space: the distance between state-action pairs depends upon the actions that the agent takes in a state. State-action pairs can now be differentiated; some actions in a state are more similar than others. This technique is capable of capturing smoothness in state-action space, which will not necessarily happen when copying basis function created in state space.

This approach creates fewer basis functions because it does not require saving basis functions that are copied or the extra weights for these basis functions. This is especially

important in domains with a large number of actions and domains where the number of actions available in each state varies significantly. Embeddings created using these graphs are also able to differentiate between actions when several actions with different costs lead from state s to state s'. In state graphs these differences cannot be modeled and would be averaged or lost. While fewer basis functions are created, the graphs will be larger, thus the eigen decomposition will be more expensive. If the agent can reuse these basis functions many times during its lifetime, this initial expense should be worthwhile.

4.4 General Analysis of State-Action Graphs

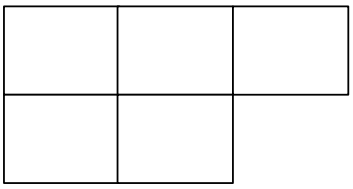
In this section, we perform three different types of analyses on the state-action graph. The first analysis describes the relationship between state-action graphs and state graphs. The second provides a general argument that Q-functions will be smooth functions over state-action graphs, and the third analyzes the Q-learning update rule for state and state-action graphs.

4.4.1 Relationship Between State-Action Graphs and State Graphs

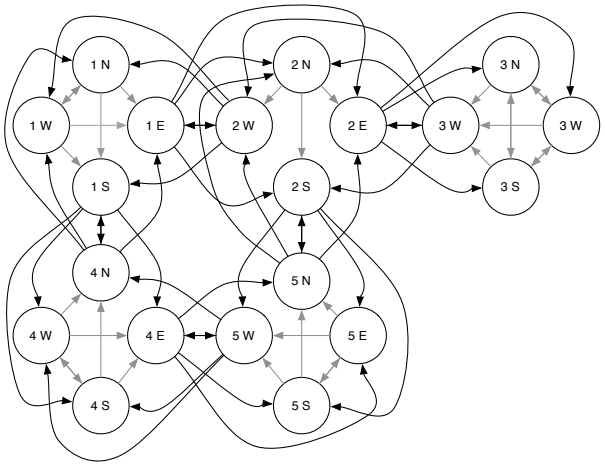
A state-action graph is a special type of graph where each vertex represents multiple types of variables. The state and state-action graphs can be viewed as two models of a corresponding underlying process. State-action graphs are strictly more general than state graphs, and state graphs can be constructed from state-action graphs. Additionally action graphs, graphs where actions are the vertices of the graph, could also be created.

In order to show this, we use the small deterministic five state gridworld in Figure 4.3 as an example. In this domain, the agent can take four actions: north, east, south, and west.

Given a state-action graph $G_{sa} = (V_{sa}, E_{sa}, W_{sa})$, the corresponding state graph $G_s = (V_s, E_s, W_s)$ can be easily constructed. V_s can be created from V_{sa} by selecting the unique set of states that create the state-action pairs. E_s can be created from E_{sa} such that any two



(a) Small, discrete five state grid-world.



(b) The corresponding state-action graph. Each edge has a weight of 1.

Figure 4.3. Example to demonstrate the relationship between state and state-action graphs.

nodes s_1 and $s_2 \in V_s$ are connected if there exist two nodes $(s_1, a_i), (s_2, a_j) \in V_{sa}$ with an edge between (s_1, a_i) and (s_2, a_j).

In our example, the unique set of states that create the state-action pairs in V_{sa} are $\{1\ 2\ 3\ 4\ 5\}$; these will be the vertices for the state graph V_s. There will be an edge between vertex 1 and vertex 2 because $(1, E)$ and $(2, W)$ are connected in the state-action graph. However, vertex 1 and vertex 3 will not have an edge between them because no state-action pair for either state 1 or state 3 that are connected in the state-action graph.

Each entry in the weight matrix W_{sa} of the state-action graph provides the weight of an edge between a state-action pair (s_1, a_1) to another state-action pair (s_2, a_2). W_s could be easily created by setting $W_s(s_1, s_2) = 1$ if an edge exists between the vertices and 0 otherwise. An alternate approach to creating W_s is through marginalization:

$$W_s(s_1, s_2) = \sum_{a_i \in A(s_1)} \sum_{a_j \in A(s_2)} W_{sa}\Big((s_1, a_i), (s_2, a_j)\Big)$$

where $W_{sa}\Big((s_1, a_i), (s_2, a_j)\Big) = 0$ if an edge does not exist between (s_1, a_i) and (s_2, a_j).

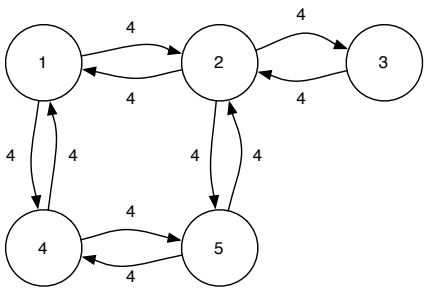


Figure 4.4. State graph generated from the state-action graph.

Figure 4.4 shows the state graph generated from the state-action graph. Since the domain is deterministic, each edge of the state-action graph will have a weight of one. In the state graph $W(1,2) = 4$. This is because for state 1, the state-action pair (1,E) has four edges extending to the four state-action pairs of state 2, and no other state-action pairs of state 1 connect to the state-action pairs of state 2.

We can also calculate the graph transition probability of two states from the state-action graph as well:

$$\mathcal{P}_s(s_1, s_2) = \frac{\sum_{a_i \in A(s_1)} \sum_{a_j \in A(s_2)} W_{sa}\Big((s_1, a_i), (s_2, a_j)\Big)}{D_{s_1}},$$

where $D_{s_1} = \sum_{a_i \in A(s_1)} \sum_{(s_j, a_j)} I(s_1 \neq s_j) W_{sa}\Big((s_1, a_i), (s_j, a_j)\Big)$. $\sum_{(s_j, a_j)}$ indicates a summation over the vertices in the state-action graph and $I(s_i \neq s_j)$ is the indicator function that returns one when $s_i \neq s_j$ and 0 otherwise. In our example $\mathcal{P}_s(1,2) = 4/8 = 1/2$.

Every entry in W_s is a linear combination of a subset of entries in W_{sa}. L_s can be represented in terms of W_{sa}. The graph Laplacian of two vertices of the state graph can be computed from the state-action graph in the following way:

$$L_s(s_1, s_2) = \begin{cases} D_{s_1} & \text{if } s_1 = s_2, \\ -\sum_{a_i \in A(s_1)} \sum_{a_j \in A(s_2)} W_{sa}\Big((s_1, a_i), (s_2, a_j)\Big) & \text{if } s_1 \text{ and } s_2 \text{ are adjacent in } G_s, \\ 0 & \text{otherwise.} \end{cases}$$

In our example $L_s(1,1) = 8$ and $L_s(1,2) = -4$.

4.4.2 Smoothness of Q-value Functions in State-Action Space

The argument for using the eigenvectors of the graph Laplacian for value function approximation is that value functions are typically smooth on the state space graph. Smoothness on the state space graph of an MDP means that if an agent can select an action that causes a transition from one state to another state then these states will have similar values. Mahadevan and Maggioni (2006) use the Sobolev norm

$$\begin{aligned} ||f||^2_{\mathcal{H}^2} &= ||f||^2_2 + ||\nabla_f||^2_2 \\ &= \sum_{v \in V} |f(v)|^2 d_v + \sum_{u \sim v} |f(u) - f(v)|^2 W(u,v) \end{aligned}$$

as a smoothness measure of functions over graphs. They state that this approach is intended for value functions that have small $\mathcal{H}_2$ norm and argue that smoothness comes from the fact that a value at a given state $V^\pi(s)$ is always a function of the values at neighboring states.

This analysis holds for value functions; however when using the state-action value function $Q^\pi(s,a)$, the basis functions are copied. The assumption is that each action is held constant and smoothness is in terms of just the states.

We can measure the smoothness of the functions over directed graphs as well. The Sobolev norm for the directed graph (Johns & Mahadevan, 2007) is defined as

$$|f||^2_{\mathcal{H}^2} = ||f||^2_2 + ||\nabla_f||^2_2 = \sum_{v \in V} |f(v)|^2 d_v + \sum_{u \to v} |f(u) - f(v)|^2 \frac{\psi_u}{d_u} W(u,v). \quad (4.1)$$

If we assume V^π has a small $\mathcal{H}_2$ norm, neighboring states will have similar values. We also know that in the state graph two vertices are connected if an action causes a transition between the corresponding states. Now consider the corresponding state-action graph. A state-action pair, (s,a) is connected to (s',a') if a causes a transition from s to s', and a' is an action that is available in s'. $V^\pi(s)$ and $V^\pi(s')$ have similar values because s and s'

are linked under π. Similarly $Q^\pi(s,a)$ and $Q^\pi(s',a')$ are linked under π. The argument is essentially the same as the argument for the state-graph. The action has merely gone from being implicitly represented on the edges to explicitly represented by the vertices.

This explicit representation is important to compactly represent the action-value function. Several actions could cause transitions between s and s'. These similarities are captured in the state-action graph and thus in the embedding. The state-action graph also represents only the actions available in a given state. Neither of these are possible when using the embedding of the state graph to represent the action-value function with copying.

4.4.3 Analysis of Updates During Learning

The difference between basis functions created directly in state-action space versus those created in state space can be understood through analysis of the parameter update rule used in Q-learning. Recall from Section 2.3 that the action value function is approximated in the following way

$$\hat{Q}^\pi(s,a|\boldsymbol{\theta}) \quad = \quad \sum_{j=1}^{k} \phi_j(s,a)\theta_j.$$

The parameter update is given by

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \alpha \cdot (r_t + \max_{a' \in A(s')} \gamma \hat{Q}_t(s',a'|\boldsymbol{\theta}_t) - \hat{Q}_t(s,a|\boldsymbol{\theta}_t)) \cdot \nabla_{\boldsymbol{\theta}_t} \hat{Q}_t(s,a|\boldsymbol{\theta}_t). \tag{4.2}$$

For linear function approximators

$$\nabla_{\boldsymbol{\theta}_t} \hat{Q}_t(s,a|\boldsymbol{\theta}_t) = \phi(s,a).$$

If Φ is a set of basis functions originally constructed on the state space and then copied for each action, $\phi(s,a)$ will be a vector of mostly zeros. We can formally write the extension of $\phi(s)$ to $\phi(s,a)$ as a Kronecker product

$$\phi(s, a) = e_I(a) \otimes \phi(s),$$

where $e_I(a)$ is the unit vector corresponding to the index of action a. For example, $e_I(a_1) = [1, 0, \ldots, 0]^T$. Since $\phi(s, a)$ is the k length embedding of the state-action pair, only the portion of the vector that corresponds to action a will contain the $\frac{k}{|A|}$ values from the state basis functions. This means only the parameters corresponding to the set of features for action a will be updated. Generalization across actions cannot occur.

If Φ is a set of basis functions constructed directly on the state-action space, $\phi(s, a)$ may be a dense vector of length k. Parameters for all state-action pairs can potentially be updated. Generalization across states and actions can occur simultaneously.

4.5 Demonstration Using Four Room Gridworld

To illustrate this technique, we use a four room gridworld shown in Figure 4.5 (Sutton et al., 1999). This domain consists of 169 states of which 104 are free states (states that

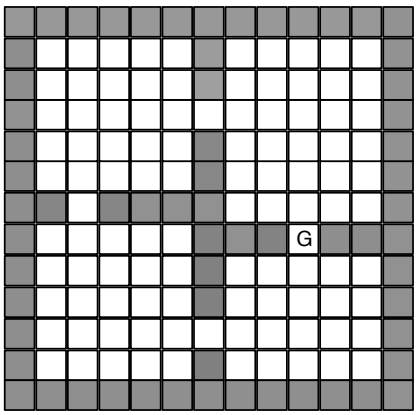
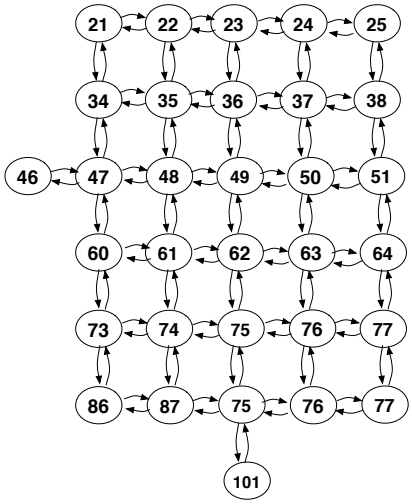


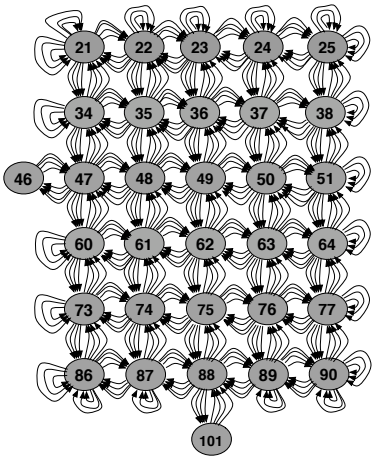
Figure 4.5. Four room gridworld.

are not a wall). In any free state the agent can perform one of four primitive actions: north, south, east or west. There is a 10% probability that an action will fail and the agent will remain in the same location. If an agent selects an action that would transition it into a

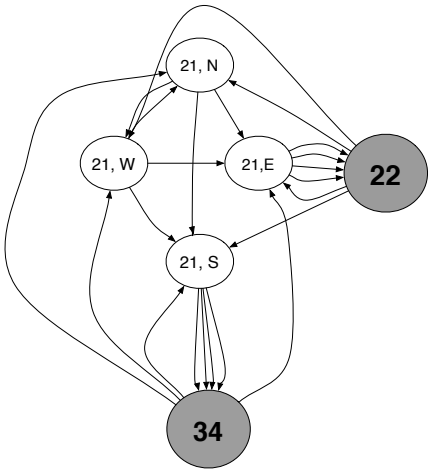
wall state, it remains in the same location. Rewards are zero on all state transitions except transitions into the goal state when the agent receives a reward of 100.



(a) State graph



(b) State-action graph.



(c) Close up of transitions associated with nodes for state action pairs for state 21.

Figure 4.6. The state action graphs created for a small room.

Figure 4.6 illustrates a portion of the graphs including only the states located in the upper right room room. Figure 4.6(a) shows the state graph. Figures 4.6(b) and 4.6(c) show the state-action graph. Figure 4.6(b) show the global topology of the graph. The global topology of the state-action graph is similar to the topology to the state graph. However,

each node in this figure represents the 4 state-action pairs for each state. Figure 4.6(c) shows the state action pairs specifically for state 21.

4.5.1 Basis Functions for the Four Room Gridworld

In order to understand the representations created from these graphs, we look at the embeddings of the graphs on the second and third eigenvector, as explained in Section 3.1.3. Figure 4.8 shows the embedding of the vertices of the graphs using the second and third eigenvectors as coordinates. The portion of the graph corresponding to the plotted subgraphs is circled. These figures show the vertices of the graphs embedded in feature space. We show only the embeddings of the combinatorial graph Laplacian since the embeddings

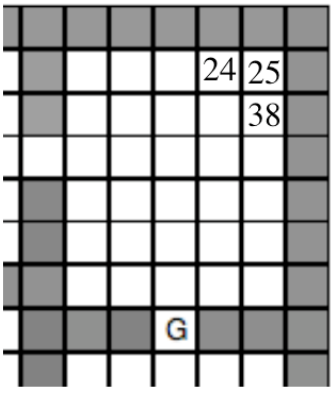
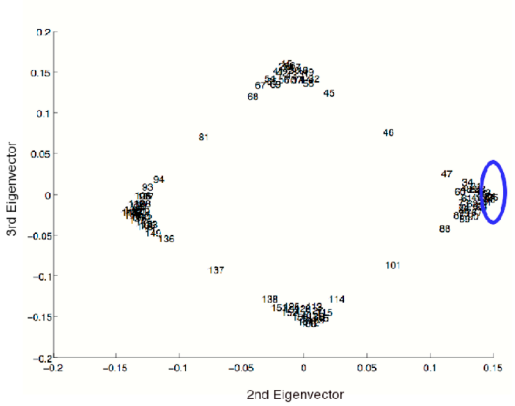


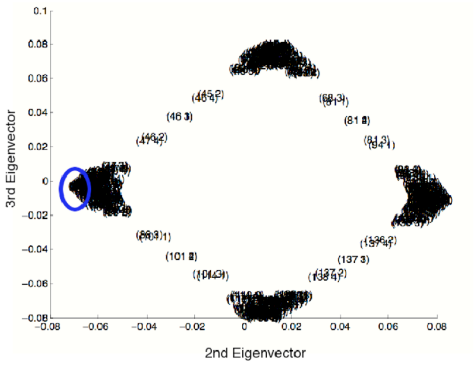
Figure 4.7. Right corner of the four room gridworld with the corner states labeled.

of the normalized graph Laplacian are similar for this domain. These figures show that both techniques group the graphs into four large clusters corresponding to the four rooms. Figures 4.8(c) and 4.8(d) provide a zoomed in view of the vertices of the graph for the states in the upper right-hand corner of the domain, shown in Figure 4.7. The circles in Figures 4.8(a) and 4.8(b) show the zoomed in area on the respective figures. Figure 4.8(c) shows the embedding of the states while Figure 4.8(d) shows the embedding of the state-action pairs. As can be seen in Figure 4.8(d), actions north (N) and east (E) in state 25 are located on the same point. This is a desirable result since state 25 is located at the top right corner

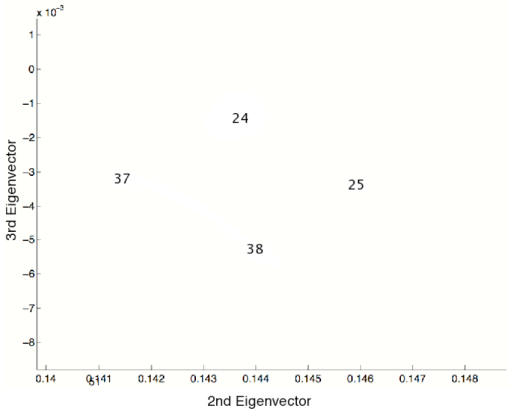
of the grid, and both actions will transition back to state 25. State-action pairs that have similar transitions are also placed near to each other: (25,S), (38,N) and (25,W), (24,E), are examples of this. This proximity in the embedding's space is highly desirable as it yields good generalization across state-action space.



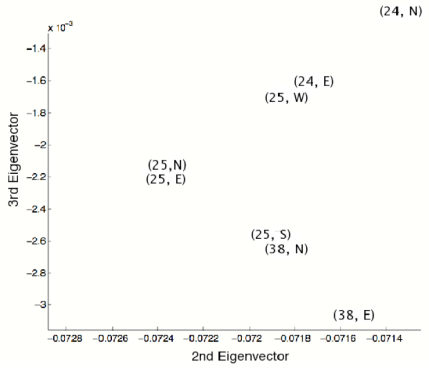
(a) Embeddings of the state graph's vertices on the 2nd and 3rd eigenvectors of the directed combinatorial graph Laplacian of the state graph.



(b) Embeddings of the state-action graph's vertices on the 2nd and 3rd eigenvectors of the directed combinatorial graph Laplacian of the state-action graph.



(c) Close up of the embeddings of the combinatorial graph Laplacian on the state graph for the right corner.



(d) Close up of the embeddings of the combinatorial graph Laplacian on the state-action graph for the right corner.

Figure 4.8. Embeddings of the four room domain on the 2nd and 3rd eigenvectors.

Figure 4.10 provides a visual comparison of the basis functions created from either the state or state-action graph of the four room gridworld. The basis functions created from the state graph are constructed by taking the combinatorial graph Laplacian of the

graph and then copying the basis functions for each action. The basis functions created from the state-action graph are constructed by taking the combinatorial graph Laplacian of that graph. Each colored square in Figure 4.9 is a valid state-action pair in the four room gridworld domain. Each state is represented by a three by three grid in this image where the north action is on the first row and second column. An explanation of this visualization

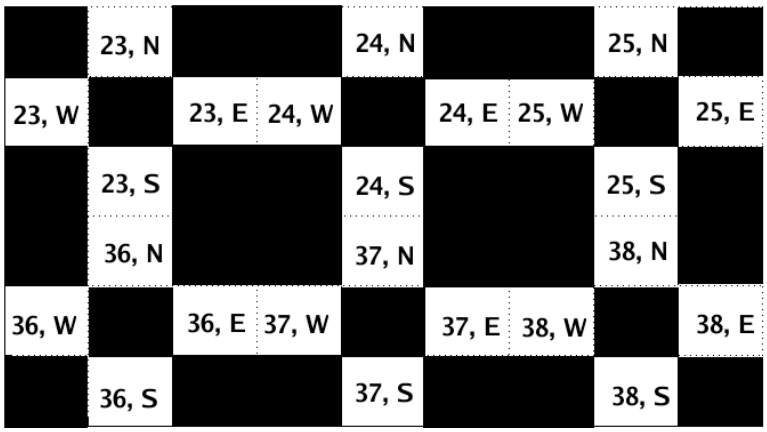
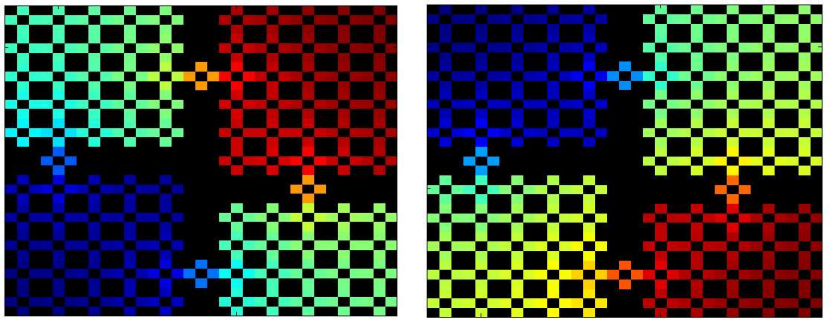


Figure 4.9. Visualization of state-action graph for the right corner of the four room gridworld.

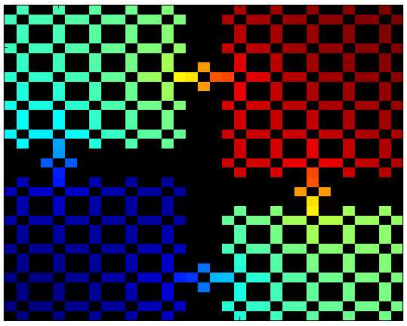
for the upper right hand corner of the four room gridworld can be seen in Figure 4.10. West and east actions are located on the second row in the first and third columns respectively. The south action is located in the third row and second column. The three by three squares are arranged to correspond to the state adjacency in Figure 4.5. The colors in the figures represent the values ranging from low, dark blue, to high, dark red, of the eigenvectors for the corresponding state-action pair. To visualize the state-action space for the state graph Laplacian we use the four basis functions in state-action space created by copying the eigenvector over the state space for each action. We do not visualize basis functions associated with the first eigenvector for either graph because this eigenvector is a constant vector.

The overall shape of the basis functions of the state and state-action graphs are similar. However, basis functions created from the state graph for a specific eigenvector have the

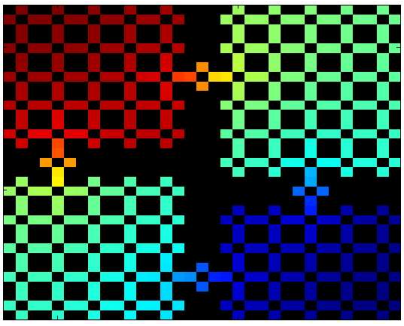


(a) 5th-8th state basis functions

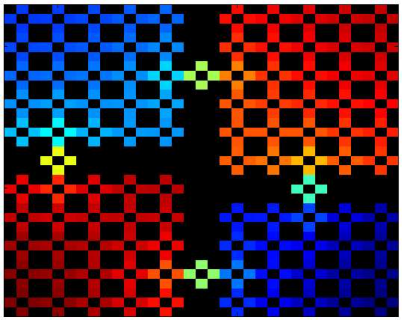
(b) 9th-12th state basis function



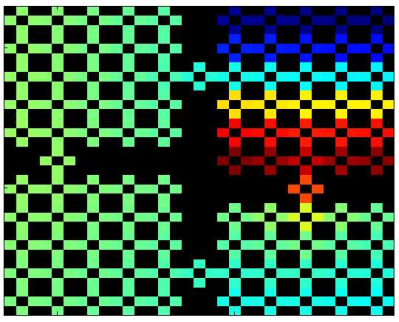
(c) 2nd state-action basis function



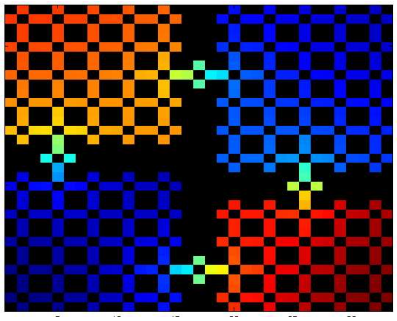
(d) 3rd state-action basis function



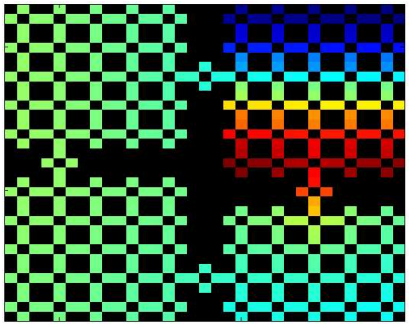
(e) 13th-16th state basis function



(f) 17th-20th state basis function



(g) 4th state-action basis function



(h) 5th state-action basis function

Figure 4.10. A visual comparison of the basis functions constructed from either the state or state-action graph on the state-action space of the four room gridworld.

same values for every action in a given state, while the basis functions created from the state-action graph are capable of having different values for different actions in a given state. The eigenvectors of the state-action graph visually have a smoother gradient than the eigenvectors from the state graph.

4.5.2 Comparison of Feature Spaces

In this section, we compare the basis functions created using eigenvectors of the graph Laplacians on the two different types of graphs. We consider three different sets of basis functions: Set 1 is created using the first eight eigenvectors with the smallest eigenvalues of the directed graph Laplacian on the state-action graph. Set 2 is the first two low order eigenvectors of the state graph Laplacian copied four times to create 8 basis functions. Set 3 is the first eight low order eigenvectors of the state graph Laplacian copied eight times to create 32 basis functions. Table 4.1 contains a summary of the set information.

Set	Graph	Number of eigenvectors	Copying	Number of basis functions
Set 1:	State-action	8	no	8
Set 2:	State	2	4 times	8
Set 3:	State	8	4 times	32

Table 4.1. Information about eigenvectors used in the comparisons.

The basis functions define a subspace within the space of functions that can be represented over the graph. It is possible to find the minimum angle between two subspaces quantitatively by projecting the two spaces onto each other (Bjorck & Golub, 1973; Wedin, 1983). The size of the angle between two subspaces, or space spanned by the basis functions, signifies how different the subspaces are. If the angle is small then the spaces are almost linearly dependent.

We can compare the angle between the subspaces spanned by the basis functions created from the state-action graph and the two sets of basis functions created on the state graph. Distance 1 is the angle between Set 1 and Set 2. These subspaces have the same number

of parameters that must be learned by the learning algorithm. However, the basis functions created from the state-action graph contain less zeros. Distance 2 is the angle between Set 1 and Set 3. These subspaces have the same number of nonzero entries in the basis functions, but the basis functions from the state graph have 4 times the number of parameters that must be examined in learning.

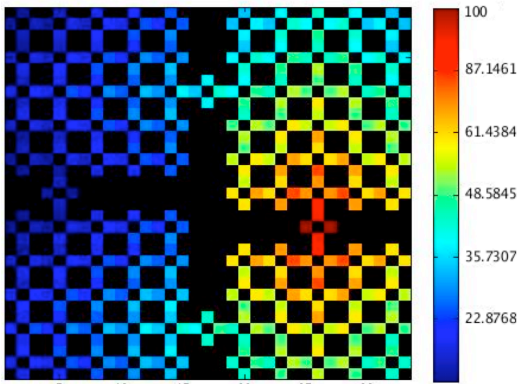
The results of our comparison can be found in Table 4.2. The angle between the subspace defined by Set 1 and the subspace defined by Set 2 is 90 degrees. The angle between the subspace defined by Set 1 and the subspace defined by Set 3 is 5.4 degrees. This indicates that while the subspace defined by Set 1 is more similar to the subspace defined by Set 3, it requires the same number of parameters as the first set of basis functions in Set 2.

Distance 1: Set 1 and Set 2	90 degrees
Distance 2: Set 1 and Set 3	5.4 degrees

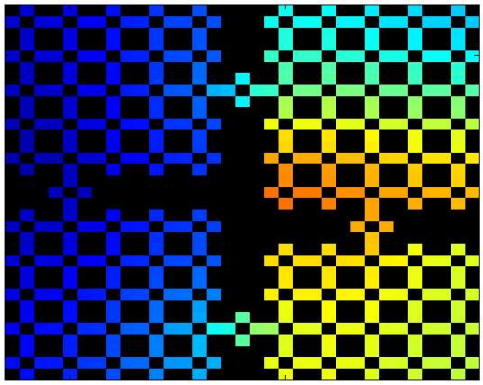
Table 4.2. Distance between subspaces induced by the eigenvectors of the graph Laplacians.

This analysis indicates that the basis functions in Set 1 and the basis functions in Set 3 have similar representational power. Figure 4.11 is a visualization of how useful the basis functions are for representing the Q-value function in this domain. We project the optimal Q-value function, displayed in Figure 4.11(a), onto the space spanned by the basis functions. Figure 4.11(b) shows the optimal Q-value function projected onto the basis functions in Set 1. Figure 4.11(c) shows the optimal Q-value function projected onto the basis functions in Set 2. Figure 4.11(d) shows the projection of the optimal Q-value function onto the basis functions in Set 3.

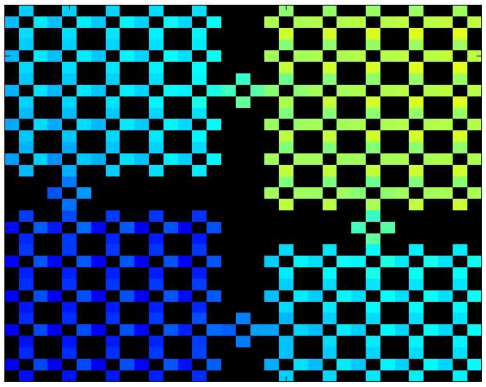
The basis functions used to represent the value function in Figure 4.11(b) and Figure 4.11(c) have the same number of parameters but the number of nonzero entries in the state-action basis functions is four times that of the state basis functions. The state-action basis functions are clearly able to more accurately represent the value function than the state



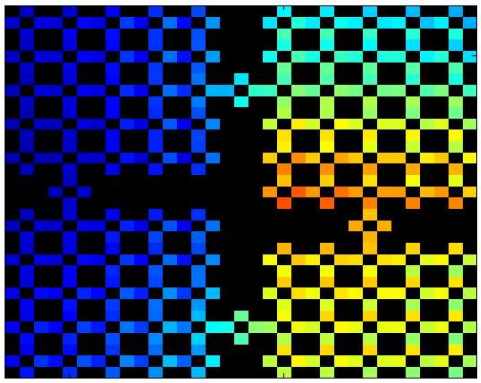
(a) The optimal Q-value function.



(b) Projection of the Q-value function onto the first 8 eigenvectors of the directed graph Laplacian of the state-action graph.



(c) Projection of the Q-value function onto the first 8 eigenvectors of the graph Laplacian of the state graph. The first 8 eigenvectors are created by copying the first 2 eigenvectors of the graph Laplacian of the state graph for each action.



(d) Projection of the Q-value function onto the first 32 eigenvectors of the graph Laplacian of the state graph. The first 32 eigenvectors are created by copying the first 8 eigenvectors of the graph Laplacian of the state graph for each action.

Figure 4.11. Visualization of the Q-function for the four room gridworld.

basis functions. The basis functions used to represent the value function in Figure 4.11(b) and Figure 4.11(d) have the same number of nonzero entries but the state basis functions have four times the number of parameters.

Figure 4.12 plots the error of the projected Q-value function. In this figure, we plot the sum of the error (or the L^1 error) of the projected Q-value function over the number of basis functions used in the projection. We show three different comparisons. The state-action basis functions are the eigenvectors of the directed combinatorial graph Laplaican of

the state-action graph. The state basis functions are the eigenvectors of the combinatorial graph Laplacian of the state graph. Since this required each eigenvector to be copied four times we also plotted the number of basis functions copied. This figure shows that the Q-function approximation created using state-action basis functions has a lower error than the approximation created using basis functions constructed from the state graph.

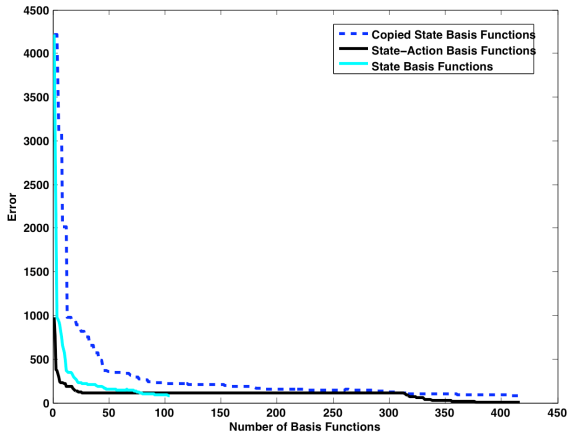


Figure 4.12. A comparison of the error of the projected Q-function. The error is the sum of the error between the optimal Q-function and the Q-function projected onto the set of basis functions.

The visualization in Figure 4.11, measures of distance between the subspaces, and error analysis of the projected Q-function all provide an intuition of how the state and state-action basis functions are related. The state and state-action basis functions perform similarly in their ability to represent the value function. However during learning, four times the number of parameters will be required to be learned for the state bases extended to the state-action bases, which may substantially slow learning.

4.5.3 Smoothness Comparison

The Sobolev norm is a measure of the smoothness of a function over a graph. In this section, we examine the smoothness of the optimal value function V over the state graph and the optimal action-value function Q over the state-action graph in the four room gridworld

problem. Since the Sobolev norm of the directed graph Laplacian, defined in Equation 4.1, is normalized by the invariant distribution it cannot be directly compared to the Sobolev norm of the undirected graph defined in Equation 3.1. We compare both approaches using the Sobolev norm of the directed graph Laplacian. The Sobolev norm of the value function V over the state graph is defined as

$$\begin{aligned} ||V||^2_{\mathcal{H}^2} &= ||V||^2_2 + ||\nabla_V||^2_2 \\ &= \sum_{(s_1)\in V_s} |V(s_1)|^2 d_{(s_1)} + \sum_{(s_1)\to(s_2)} |V(s_1) - V(s_2)|^2 . \frac{\psi_{(s_1)}}{d_{(s_2)}} W(s_1, s_2). \end{aligned}$$

The Sobolev norm of the action value function Q over the state-action graph is defined as

$$\begin{aligned} ||Q||^2_{\mathcal{H}^2} &= ||Q||^2_2 + ||\nabla_Q||^2_2 \\ &= \sum_{(s_1,a_1)\in V_{sa}} |Q(s_1, a_1)|^2 d_{(s_1,a_1)} + \\ &\quad \sum_{(s_1,a_1)\to(s_2,a_2)} |Q(s_1, a_1) - Q(s_2, a_2)|^2 \frac{\psi_{(s_1,a_1)}}{d_{(s_2,a_2)}} W((s_1, a_1), (s_2, a_2)). \end{aligned}$$

We first compare the Dirichlet sums, the second term of the Sobolev norm. Table 4.3 shows the differences between the Dirichlet sum for the appropriate value function over the three different types of graphs. The Q-function over the directed state-action graph has the smallest value.

V over State Graph:	0.4620
Q over Undirected State-Action:	0.5528
Q over Directed State-Action Graph:	0.4206

Table 4.3. Dirichlet Sum Comparison

We next compare the Sobolev norm for the four room gridworld for the state graph, the undirected state-action graph, and the directed-state action graph. Table 4.4 displays the

Sobolev norm for each of each graph normalized by the number of vertices in the graph. The results demonstrate that the undirected state-action graph does not capture smoothness properties of the Q-function as effectively as the directed state-action graph. Additionally our results show that the Q-function over the state-action graph is smoother than the V function over the state graph.

V over State Graph:	791.975
Q over Undirected State-Action:	1461.1
Q over Directed State-Action Graph:	627.772

Table 4.4. Sobolev Norm Comparison

4.6 Experimental Evaluation

In the previous section we analyzed the potential of the state-action basis functions for Q-value function approximation. In this section, we experimentally evaluate these basis functions and compare them to basis functions created on the state graph as well as more traditional approaches.

4.6.1 Learning Action-Value Functions Using State-Action Basis Functions

Figure 4.13 shows the pseudo-code for our experimental approach. The agent performs an initial period of exploration where it collects samples. These samples are then used to build basis functions, and the basis functions are all used during learning.

The learning algorithm we use in this section is Q(λ)-learning (Sutton & Barto, 1998) to learn a policy that maximizes the agent's return. We selected Q(λ) because it is an off-policy learning technique and we will use the samples collected for basis function construction for learning. The learning algorithm uses an ϵ-greedy policy for action selection

and accumulating eligibility traces. Traces are set to zero when a random action is taken. The update rule for the parameters is given as:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \delta_t \boldsymbol{e}_t, \tag{4.3}$$

where

$$\delta_t = r + \gamma \max_{a' \in A(s')} \hat{Q}_t(s', a'|\boldsymbol{\theta}_t) - \hat{Q}_t(s, a|\boldsymbol{\theta}_t),$$

$$\boldsymbol{e}_t = \gamma \lambda \boldsymbol{e}_{t-1} + \nabla_{\boldsymbol{\theta}_t} \hat{Q}_t(s, a|\boldsymbol{\theta}_t), \text{and } \boldsymbol{e}_0 = \mathbf{0}.$$

4.6.2 Experiments On The Four Room Gridworld

We first consider a learning problem in the four room gridworld domain. The agent must use the primitive actions (north, east, south, west) to learn to reach the goal. We allow the agent to explore the environment using a policy that selects the least frequently used action in a state. This exploratory period was only performed once and the set of samples were reused for all experiments in this domain. We used 2000 episodes with 50 steps per episode. During the exploration period the agent's initial state is randomly selected from the set of states that are not a wall.

We performed experiments to compare the performance of using a state-action graph versus a state graph. For these experiments we used the normalized graph Laplacian. We systematically varied the number of basis functions used during function approximation. In experiments using state graphs we varied the number of basis functions from 40 to 200 in steps of 40. In experiments using the state-action graphs we varied the number of basis functions from 50 to 200 in increments of 50. The result of each experiment was averaged 200 times and each experiment consisted of 200 episodes.

SA-RPI Algorithm $(\mathcal{D}, \gamma, \epsilon, k, \pi_0)$:
// $\mathcal{D}$: Source of samples (s, a, r, s')
// γ: Discount factor
// ϵ : Stopping criteria
// k: Number of basis functions
//π_0: Initial policy specified as a weight vector w_0

1. **Sample Collection:** Generate a set of samples $\mathcal{D}$, which consists of a state, action, reward, and next state, (s, a, r, s'). The samples are created using a series of exploratory trajectories using π_0, where π_0 selects the least frequently used action in any state. During sample collection the agent was placed in a random location at the beginning of an episode.
2. **Representation Learning:**
   (a) Build a directed weighted graph G_{sa} from $\mathcal{D}$ where V_{sa} is the set of state-action pairs, E is the edge set, and W is the weight matrix constructed using the off-policy approach in Figure 4.2.
   (b) Calculate the k lowest order eigenfunctions of the (combinatorial or normalized) graph Laplacian operator on G. These k eigenvectors make up the basis functions ϕ.
      i. Form the directed Laplacian per Equation 3.9 or 3.10.
      ii. Calculate ϕ by computing the eigenvectors of the directed Laplacian.
3. **Control Learning Phase**:
   Use Q(λ)-learning as the parameter estimation method to learn a policy π using the parameter update in Equation 4.3.

Figure 4.13. RPI Framework for learning representation and control using state-action graphs.

Figure 4.14 compares the number of steps taken by the agent to reach the goal when the graph is created on the state and state-action graphs. When examining the results, it is important to recall that the basis functions for a state graph are the eigenvectors of the graph Laplacian copied $|A|$ times. $|A|$ is the number of actions. In the four room domain there are four actions. This means that one eigenvector of the graph Laplacian on the state graph will be four basis functions for the learning algorithm. In this figure, we report the number of basis functions not the number of unique eigenvectors. We also varied α, selecting the setting of α that provided the best learning performance without divergence. The experiments with state-action basis functions plotted in Figure 4.14 both had $\alpha = .1$.

The state basis function experiment had $\alpha = .05$. λ and γ were both set to .9 for all experiments and $\epsilon = .1$.

The best performance we observed using basis functions built from the state graph required using 200 basis functions. We observed similar convergence when using 50 state-action basis functions. When we used 100 state-action basis functions, we found that the agent achieved better performance significantly quicker.

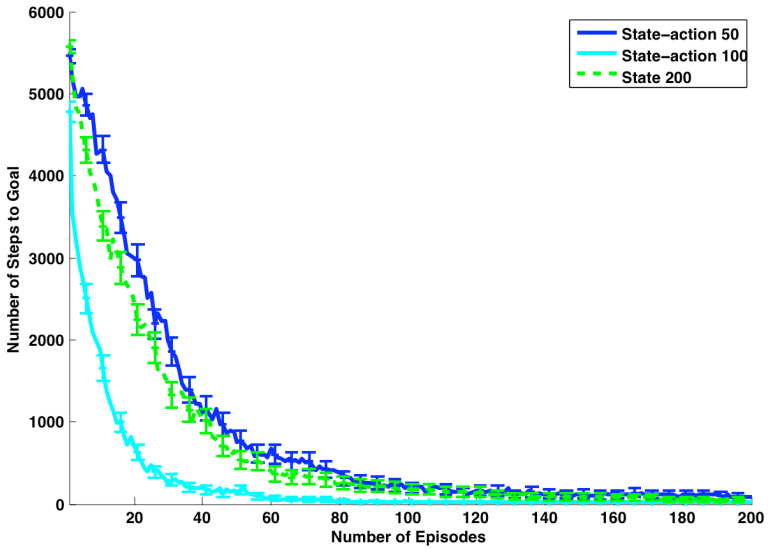


Figure 4.14. Results for learning in the four room gridworld.

4.6.3 Mazeworld

We also examined performance of state-action basis functions in a maze world, pictured in Figure 4.15. This domain contains a greater number of obstacles than the four room gridworld, thus many states will have multiple actions with a similar effect. The domain has 400 states and 4 actions, corresponding to the cardinal directions, are available in each state. There is a 10% probability that an action will fail; when this occurs the agent remains in the current state. If an agent hits a wall, it remains in the same location. Rewards are -1 on all state transitions except transitions into the goal state where the agent receives a reward of 100.

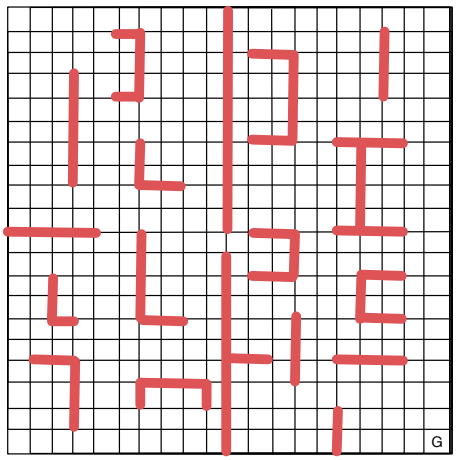


Figure 4.15. The mazeworld domain.

Our experiments in this domain required the agent to learn to navigate from the upper leftmost square to the goal state in the right bottom corner. To create the samples we used a policy that selected the least frequently used action in each state. The samples were collected using 2000 episodes with 100 steps per episode. The agent started each episode in a random state. We performed this exploration only once. We built the state-action graph using off-policy graph creation and used the eigenvectors of the normalized graph Laplacian. We used the approach in Figure 4.13 with Q(λ)-learning. For all experiments $\gamma = 0.9$, $\epsilon = 0.1$, $\alpha = .1$, and $\lambda = .9$. Each episode was halted after 5000 steps. The agent started each episode in the learning phase in the top left corner of the domain.

We experimentally varied the number of state-action basis functions used from 100 to 1000 in increments of 100. We varied the number of state basis functions from 400 to 1600 in increments of 200. Each trial was run for 200 episodes and results are averaged over 50 trials. Figure 4.16 shows that faster convergence is achieved when using state-action graphs and that fewer basis functions are required to achieve this performance.

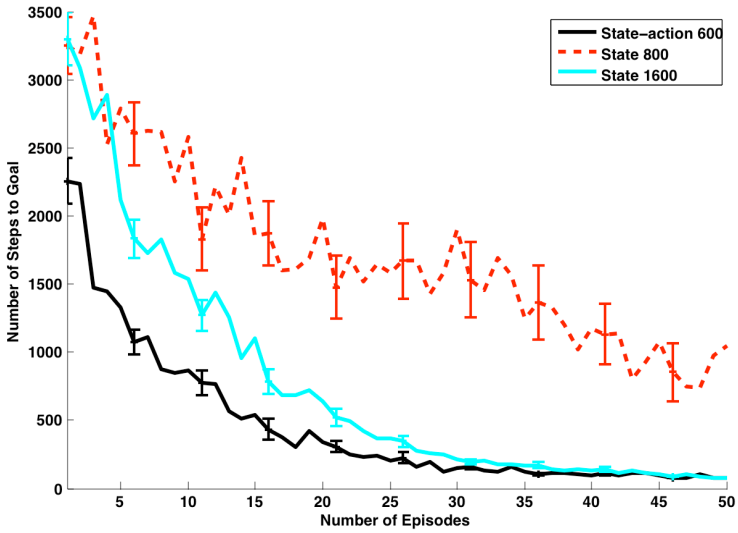


Figure 4.16. Results for learning in the maze world.

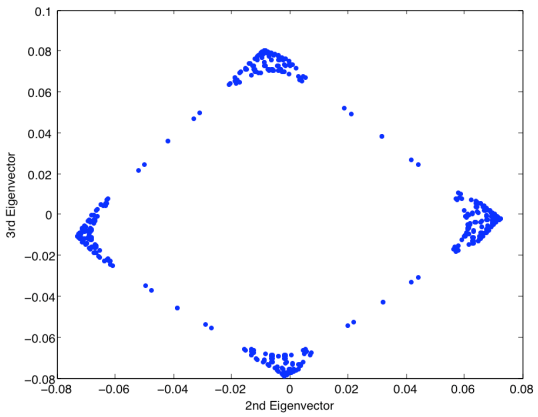
4.6.4 Graph Weighting Comparison

In the previous set of experiments we assumed that W was an approximation of P. In this section, we experimentally demonstrate the effects of different weightings on the state-action graph. Table 4.5 displays the three types of graph weightings we consider. In this table, u and v are vertices that correspond to (s, a) and (s', a'). $count(u)$ is the number of times that action a was selected in state s in the set of samples $\mathcal{D}$. $count(u, v)$ is the number of times action a was selected in state s and a transition to s' was observed and a' is one of the available actions in s'. Weighting 1 is the weighting we used previously in this chapter and can be viewed as an MLE approximation of P. Weighting 2 is can be seen as a frequency count of the transition that is not normalized. Weighting 3 is the simplest weighting in which all edges have a weight of 1.

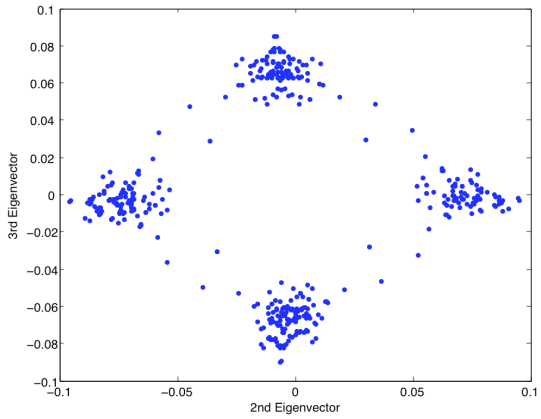
Weighting 1	$W(u, v) = \frac{count(u,v)}{count(u)}$
Weighting 2	$W(u, v) = count(u, v)$
Weighting 3	$W(u, v) = 1$ if edge else 0

Table 4.5. Weightings used for state action graphs

To understand the effects of the weightings, we first visualize the changes in the basis functions when using Weighting 1 and Weighting 3. In this visualization, we assume the agent has access to perfect information about the domain. Weighting 1 will actually be P for this visualization. The second weighting will treat all of the neighboring vertices of a specific vertex as equivalent. Weighting 1 will penalize transitions that are less frequent. Figure 4.17 shows a comparison of the effects of the two weighting approaches on the embedding of the vertices of the state-action graph on the 2nd and 3rd eigenvectors of the directed combinatorial graph Laplacian. In both embeddings the state action pairs are separated according to which room they belong. However, the embedding using Weighting 3 is more scattered than the embedding using Weighting 1. When estimating Weighting 1 from data, we expect that the embedding will fall somewhere between these two embeddings depending upon the accuracy of the estimate.



(a) Embedding of state-action graph on the 2nd and 3rd eigenvectors of the directed combinatorial graph Laplaican when Weighting 1 was used.



(b) Embedding of the state-action graph on the 2nd and 3rd eigenvectors of the directed combinatorial graph Laplaican when Weighting 3 was used.

Figure 4.17. A visual comparison of the state-action graph embedding of the four room gridworld for the two different weighting techniques.

To understand the effect of the different weightings on learning, we experimentally evaluated this approach on the four room gridworld using the same set up as in Section 4.6.2. (In this experiment, we use the samples to calculate the weights upon the graph.) In this experiment, $\alpha = .1$ for all experiments, except Weighting 3 with 50 basis functions.

We found these experiments frequently diverged when $\alpha = .1$ and set $\alpha = .05$. Figure 4.18 shows the results of the different weightings averaged 200 times with standard error bars plotted every 5 episodes. The weightings do not have a large effect upon the learning results. The only really large difference we see is when using Weighting 3 with 50 basis functions, but this is likely due to the fact it was necessary to decrease α.

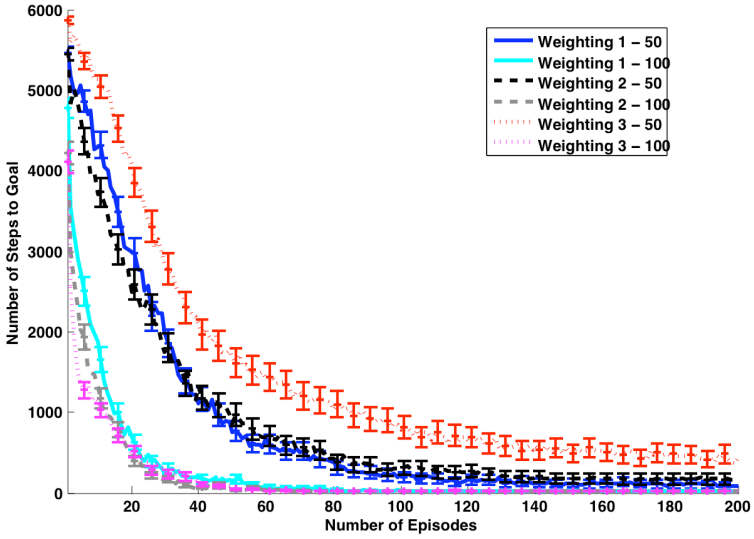


Figure 4.18. Results comparing the two weighting approaches in the four room gridworld.

4.6.5 Graph Laplacian Comparison

All other experiments in this chapter have been performed using basis functions from the normalized graph Laplacian. In this section, we compare the difference in results in learning between basis functions created using the directed normalized Laplacian and the directed combinatorial Laplacian. Once again, we use the same experimental set up as in Section 4.6.2. All experiments were run with $\alpha = .1$. Figure 4.19 shows the results of learning averaged 200 times with standard error bars plotted every 5 episodes. We do not see a significant change in the learning experiments between using the different types of

directed graph Laplacian when using 100 basis functions. However, when using 50 basis functions we see that the normalized Laplacian performs somewhat better.

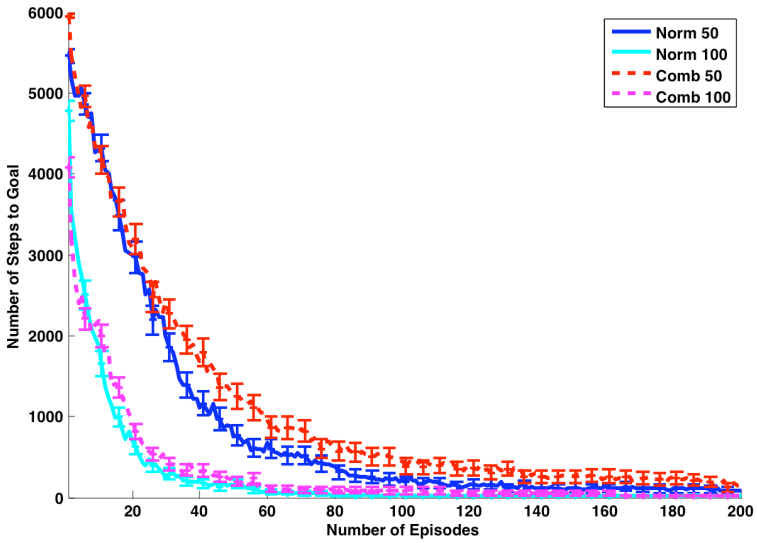


Figure 4.19. Results comparing the normalized and combinatorial Laplacians in the four room gridworld.

4.6.6 Directed Versus Undirected Graph Comparison

The last set of experiments that examine the construction of the basis functions in state-action space compare using a directed graph versus an undirected graph. Experiments with the directed graph were run with $\alpha = .1$ and $\alpha = .01$ in experiments with the undirected graph. Figure 4.20 shows the results of learning averaged 200 times with standard error bars alloted every 5 episodes. Learning performance severely degrades when basis functions constructed on the undirected graph are used. The undirected graph will have a substantial number of erroneous extra edges. Twice as many basis functions are required to achieve comparable learning performance.

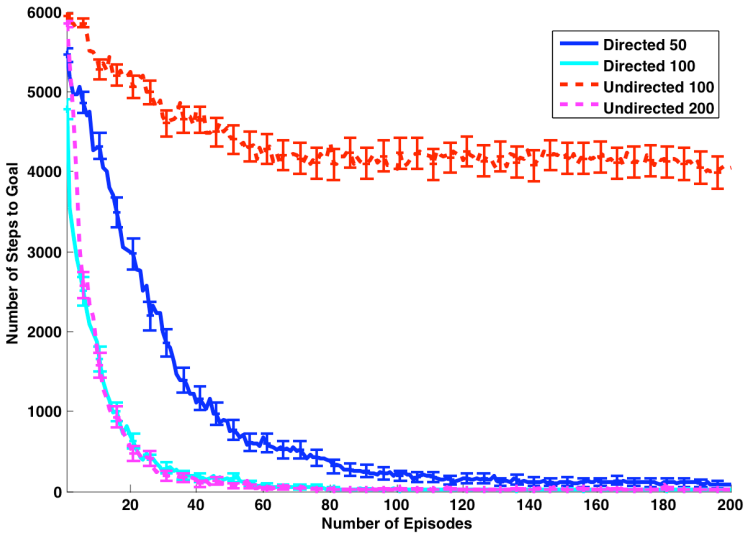


Figure 4.20. Results comparing the directed and undirected graph Laplacian on state-action graphs.

4.7 Comparison to Alternate Approaches for Basis Function Construction

In this section, we compare our approach to other basis functions approaches in the literature. We specifically compare to radial basis functions (RBFs) and Geodesic Gaussian Kernels (GGKs) (Sugiyama et al., 2007). Figure 4.21 shows a comparison of our approach with these basis functions. We also discuss the differences between our approach and that of Bellman error basis functions (BEBFS), a type of policy-specific basis function. First we give a brief overview of the other basis function types and discuss how we performed the experiments involved with each type of basis function.

4.7.1 Radial Basis Functions

RBFs are created by tiling the state space with Gaussians. Each basis function is one of the Gaussians. The Gaussians have a mean, located at one of the states, and a fixed variance σ. The value of a state for basis function i is written as

$$\phi_i(s) = \exp(-\frac{D(s, c_i)}{2\sigma^2}), \tag{4.4}$$

where c_i is the center of the i-th Gaussian and σ is the variance parameter and $D(s, c_i)$ is the distance between s and c_i. Typically $D(s, c_i)$ is the Euclidean distance $D(s, c_i) = ||s - c_i||^2$. In our experiments we will define $D(s, c_i)$ to be the Manhattan distance over the grid. In order to transform basis functions from state space to state-action space, the RBFs are copied for each action.

Placement of the Gaussians is up to the designer. In our experiments we tile 50 Gaussians over the state space uniformly. Our decisions in terms of distance metric and placement incorporate our knowledge of the state space. If we were uncertain about the domain, we may have just randomly placed the Gaussians over the state space and assumed Euclidean distance. Although we have included some of our knowledge about the domain, RBFs cannot capture information about obstacles and will still perform poorly.

We tried several settings of σ. Specifically we ran experiments with $\sigma = 1, 2, 3$ and 5. Generally RBFs performed poorly, and it was difficult to find a good setting of α for any setting of σ. This is mainly due to the fact that the basis functions use euclidean distance. Thus states that should have different values are actually represented as similar in the basis functions. For example, states separated by a wall are likely to be represented as similar even though the agent cannot transition between them.

4.7.2 Geodesic Gaussian Kernels

Geodesic Gaussian Kernels (GGKs) (Sugiyama et al., 2007; Sugiyama et al., 2008) are an approach to basis function construction for value function approximation. This work extends traditional RBFs to be constructed over the state space manifold. The approach builds basis functions by placing Gaussian kernels over the MDP state graph.

In the GGK approach the distance between states is calculated using the shortest paths on the graph, changing Equation 4.4 to

$$\phi_i(s) = \exp(-\frac{SP(s, c_i)^2}{2\sigma^2}) \tag{4.5}$$

where $SP(s, c_i)$ is the shortest path on the graph from state s to state c_i, which is once again the center of the i-th Gaussian. Shortest paths on graphs can be calculated using the Dijkstra algorithm (Dijkstra, 1959).

Sugiyama et al. (2007) propose an approach to extending GGKs from the state space to state-action space using a "shift" approach. Rather than copy the basis functions for each action, the feature of a state is the linear combination of the features of the potential next state for a state-action pair. This approach is also employed by Gärtner et al. (2003) and Driessens et al. (2006). Sugiyama et al. (2007) report that this approach works best in deterministic domains or domains where P is known. This approach will construct a parameter vector of the same size as the simple copying mechanism. Since P is not known in our experiments and our domain is not deterministic, we perform experiments with the simple copying mechanism.

Both GGKs and basis functions based on the graph Laplacian measure distances on the graph. GGKs and RBFs are both examples of local basis functions. This allows them to approximate value functions that are locally smooth but not globally smooth. However, the shortest path distance metric is easily susceptible to erroneous edges in the graph.

Two parameters must be specified for GGKs: the centers, c_i and the standard deviation σ. The placement of the the Gaussians is important for performance. We performed preliminary experiments that indicated that learning performance when the Gaussians were placed randomly varied significantly, which corresponded to intuition from the literature. For the comparison experiments, we placed the Gaussians uniformly over the state space graph. It is important to note that GGKs do not constitute an entirely learned basis since there is no automated algorithm to determine the placement of the Gaussians. While uniform placement is well defined for spaces like the state space of a grid world, it is difficult

to imagine what uniform placement may be on more complex graphs. It may be possible to adapt approaches that automatically tune RBF placement for this approach.

We tried several settings of σ: $\sigma = 1, 2, 3, 5,$ and 10. We found that smaller values of σ performed better on the four room gridworld with $Q(\lambda)$-learning. Specifically we will report results with $\sigma = 2$, although the results for $\sigma = 1$ were similar. Larger values of σ required significantly lower learning rates in our experiments.

4.7.3 Bellman Error Basis Functions

Bellman error basis functions (BEBFs) or Krylov basis functions were introduced by (Poupart & Boutilier, 2002; Keller et al., 2006; Parr et al., 2007; Petrik, 2007). In this approach, basis functions are created by explicitly using a measure of the error in value function approximation called the Bellman residual $T^{\pi}(V) - V$. The Bellman residual is approximated over the set of samples using the following sample-based TD error:

$$\hat{Q}_k(s', \pi_k(s')) + r - \hat{Q}_k(s, a).$$

These basis functions can easily be constructed directly in state-action space since each basis function is an estimate of the error of the current estimate of the action-value function. BEBFs have been exclusively used with least squares techniques. The approach is well suited to policy iteration techniques since the policy is held fixed and evaluation is done over the entire set of collected samples, and then the updates to the basis functions and parameters are computed.

BEBFs have not yet been extended to TD methods, which present interesting challenges. Updates to the value function are often small local updates, which means that basis functions constructed from the Bellman error will tend to be similar to delta functions. Additionally BEBFs are currently only appropriate for on-policy learning and would need to be extended for off-policy approaches.

4.7.4 Discussion of the Comparisons

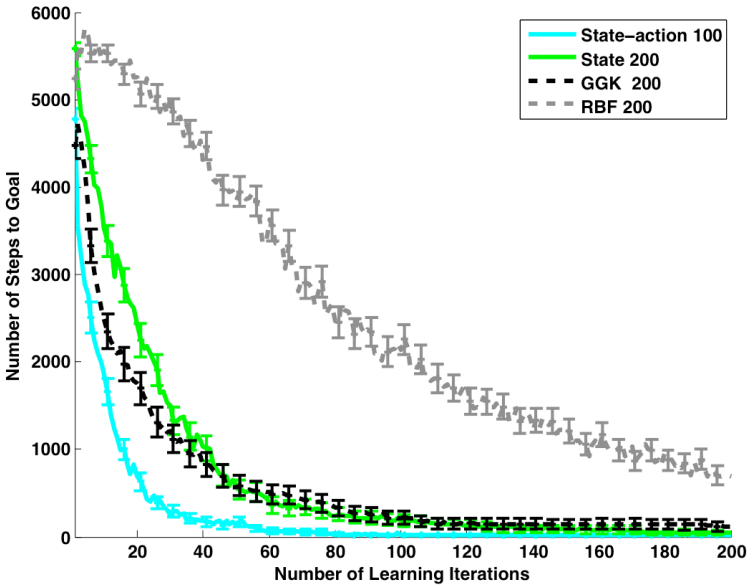


Figure 4.21. Results comparing different basis function approaches in the four room grid-world.

Our comparison shows that RBFs perform quite poorly. This is expected since the basis functions constructed using RBFs cannot take obstacles into account. GGKs and state graph Laplacian basis functions perform quite similarly. This is somewhat unsurprising in this domain since both construct basis functions with distances between states calculated on the state space and the Q-function is globally smooth. However, both of these approaches must still copy their basis functions for each action, and thus have larger parameter vectors. The state-action graph outperforms all of the approaches. If GGKs were placed on the state-action graph we believe they would achieve similar performance as long as the placement was correct. However, it is unclear how to place Gaussians over state-action graphs "uniformly". An automated approach for placing the Gaussians would most likely be helpful (McLoone et al., 1998; Moody & Darken, 1989; Sanchez, 1995; Karayiannis, 1999; Haykin, 1999; Gonzalez et al., 2003; Lazaro et al., 2003).

4.8 Conclusion

In this chapter, we defined an approach for automatically constructing basis functions for action-value functions. We introduced state-action graphs and describe how basis functions are constructed using the graph Laplacian. We described two approaches to creating these graphs: on-policy and off-policy graph creation.

Our analysis shows that basis functions created on the state-action graph are closely related to those created on the state graph. However, our approach performs better because actions are explicitly incorporated into the bases.

We experimentally evaluated the performance of these basis functions for learning action-value functions. Our results demonstrate that basis functions created from the state-action graph significantly improve learning performance when compared to basis functions created on the state-graph. Additionally, the basis functions are fairly resilient to settings of many of the parameters in domains where actions have small local effects.

CHAPTER 5

REPRESENTATION DISCOVERY IN SEMI-MARKOV DECISION PROCESSES

Thus far, this dissertation has focused on incorporating actions into representations when the actions take a single time step. A significant advance in RL has been the introduction of temporal abstraction frameworks and hierarchical learning algorithms (Barto & Mahadevan, 2003). These frameworks allow the agent to employ temporally-extended actions that allow it to make decisions at different time scales. Humans frequently employ such techniques and do not consciously plan at lower levels, such as muscle movements. For example, consider the task of baking a cake. This task could be summarized into the following subtasks: "get the ingredients", "combine", and "bake". This task itself could be part of a high level task such as planning a birthday party. The ability to simultaneously reason at multiple scales greatly improves the applicability of RL algorithms.

In this chapter, we assume the agent has access to skills or macro-actions. We use semi-Markov decision processes (SMDPs) (Puterman, 1994) as the underlying model. SMDPs are a generalization of MDPs in which actions are no longer assumed to take a single time step and may have varied durations. An SMDP is defined as a tuple (S, A, P, R). All components have the same definition as in an MDP except the transition probability function P and the reward function R. S is the set of states, and A is the set of actions the agent may take at each decision point. The transition probability function P is modified to take into account the duration of the actions. P is now a multi-step transition probability function, where $P(s', N|s, a)$ denotes the probability that action a taken in state s will cause a transition to state s' in N time steps. The reward function is also modified to take into account the duration of the action. Rewards can accumulate over the entire duration of

an action. The reward function $R(s', N|s, a)$ is the expected reward received from selecting action a in state s and transitioning to state s' with a duration of N time steps. An SMDP can be seen as representing the system at decision points, while an MDP represents the system at all times.

Much work has been done in learning value functions for SMDPs (Dietterich, 1998; Parr & Russell, 1998; Sutton et al., 1999). In this chapter, we will focus on the options framework (Sutton et al., 1999) an option, o, is defined as a tuple $\langle I, \pi_o, \beta \rangle$ where I is the initiation set of states, where the option may be initiated. π_o is the option policy, which determines how the option will select actions or other options for execution. β is the termination condition, which gives the probability of option termination after each action in the world.

Prior work in hierarchical reinforcement learning (HRL) has not explored automatic basis function construction approaches. In this chapter, we examine how basis functions can be constructed for SMDPs. We examine both state space compression and state-action space compression.

Definition 5.1 Automatic State-Action Space Basis Construction Problem in SMDPs: *Given a Semi-Markov Decision Process $M = (S, A, P, R)$, automatically construct a low-dimensional representation Φ such that the size of Φ is $|S| \times k$ or $|S||A| \times k$ where $k << |S|$ or $k << |S||A|$. Φ should be constructed such that M can be solved "accurately" in less time than an exact representation.*

Chapters 3 and 4 described techniques for automatically building basis functions for MDPs. In this chapter, we extend these approaches to SMPDs. This work is the first exploration of automatic basis function construction for SMDPs.

Much work has been done on discovering naturally useful activities in a domain. We assume that the skills are predefined or already learned using one of the various techniques in the literature. More details on skill learning can be found in the following papers Thrun and

Schwartz (1995); McGovern (2001); Hengst (2002); Şimşek and Barto (2004); Bonarini et al. (2006); Mehta et al. (2008); Şimşek and Barto (2008); Konidaris and Barto (2009).

This work can be seen as being orthogonal to our own. We assume that the skills or macro-actions are predefined. However, others have used similar fundamental techniques to those that we employ, such as spectral methods, to learn skills (Menache et al., 2002; Şimşek et al., 2005). Our approach could potentially lead to a new approach for skill discovery; however, this is currently not the focus of our work.

5.1 Graph Creation in Semi-Markov Decision Processes

Our graph construction approach for SMDPs is similar to the approach proposed in Section 4.1. However, a few modifications are necessary to account for the variable duration of the actions. We propose modifying the weight matrix W to take the duration of the temporally extended actions of the SMDP into account. This is done by weighting each option edge by the inverse of average duration of the action.

Definition 5.2 SMDP State Graph: *An SMDP state graph $G_s = (V_s, E_s, W_s)$ is defined over an SMDP M such that each vertex $v \in V_s$ corresponds to a state s such that $s \in S$. An edge exists between a vertex u and a vertex v if an action $a \in A(s)$ causes a transition between s (corresponding to u) and s' (corresponding to v). The weight matrix is constructed according to Table 5.1.*

Definition 5.3 SMDP State-Action Graph: *An SMDP state-action graph $G_{sa} = (V_{sa}, E_{sa}, W_{sa})$ is defined over an SMDP M such that each vertex $v \in V_{sa}$ corresponds to a state-action pair (s, a) such that $s \in S$ and $a \in A$. An edge exists between a vertex u and a vertex v, corresponding to (s', a'), such that action a causes a transition from s to s' and action $a' \in A(s')$. The weight matrix is constructed according to Table 5.1.*

Table 5.1 shows the weightings for the edges of graphs created on SMDPs. $W(u, v)$ indicates the weighting between two vertices of the graph. In the state-action graph, $W(u, v)$

is the weight in the state-action graph for the edge between u corresponding to (s, a) and v corresponding to (s', a'). $avetime(u, v)$ is the average duration of the action transitioning from state s to state s' using action a. $count(u)$ is the number of times the agent is in state s and selects action a. $count(u, v)$ is the number of times the agent selects action a in state s and transitions to state s' and selects action a'. In the state graph, $W(u, v)$ is the weight for the transition between state s and state s'. $avetime(u, v)$ is the average duration of the action transitioning from state s to state s'. Figure 5.1 shows the pseudo-code for the technique used for creating state-acton graphs for SMDPs using these weightings.

State-action graph	$W(u, v) = \frac{1}{avetime(u,v)} \frac{count(u,v)}{count(u)}$
State graph	$W(u, v) = \frac{1}{avetime(u,v)}$

Table 5.1. Weightings used for SMDP graphs

5.2 Demonstration Using Four Room Gridworld

We now consider the full specification of the four room domain described in Section 4.5. Two hallway macro-actions are provided in each of the four rooms. These macro-actions allow the agent to navigate from any location within one room to one of the two hallway states that lead out of that room. The macro-actions may be called from any state within a room. A hallway macro-action's policy is optimal and cannot be terminated once selected until it reaches its goal state. Hallway states do not have hallway macro-actions available to them; in these states only primitive actions are available. Figure 5.2 shows the structure for the portion of the graph for the upper right room when the agent has access to macro-actions. The graph has a similar structure as the graph in Figure 4.6(a). However, the macro-actions introduce long edges going from each state to states 46 and 101. Figure 5.3 shows the state-action pairs for state 21 when the agent has access to the hallway macro-actions. Two nodes are added for each state to represent the new state-action

SMDP State-Action Graph Creation ($\mathcal{D}$):
// $\mathcal{D}$: Source of samples (s, a, r, s', a', t)
// t is the duration of the transition between s and s' using action a.
// Creates a graph $G_{sa} = (V, E, W)$ where V is the set of state-action pairs in $\mathcal{D}$.
// u: refers to a state-action pair (s,a) found in $\mathcal{D}$
// v: refers to a state-action pair (s', a') found in $\mathcal{D}$

// $\Upsilon_{\mathcal{D}}(u)$: refers to a function that returns the number of times u is observed in $\mathcal{D}$
// $\Upsilon_{\mathcal{D}}(u, v)$: refers to a function that returns the number of times a transition between u and v is observed in $\mathcal{D}$

c is initalized to a sparse matrix of size $(S \times A) \times (S \times A)$

If using on-policy graph creation:
  $W\big((s,a),(s',a')\big) = \Upsilon_{\mathcal{D}}\big((s,a),(s',a')\big)$
  For all $(s, a, s', a', N) \in \mathcal{D}$
    $c\big((s,a),(s',a')\big) = c\big((s,a),(s',a')\big) + N$

Else for off-policy graph creation
  $W\big((s,a),(s',a')\big) = 0$
  For all $(s, a, s', a'N) \in \mathcal{D}$
    For all $a'' \in A(s')$
      $W\big((s,a),(s',a'')\big) = W\big((s,a),(s',a'')\big) + 1$
      $c\big((s,a),(s,a'')\big) = c\big((s,a),(s',a'')\big) + N$
For all $W(u, v) \neq 0$
  $W(u,v) = W(u,v)/\Upsilon_{\mathcal{D}}(u) * W(u,v)/c(u,v)$

Figure 5.1. Pseudo-Code for creating state-action graphs in SMDPs.

pairs. Transitions from these nodes will lead to the 4 state-action nodes of the hallway states associated with the transition.

5.2.1 Comparison of Basis Functions in MDPs and SMDPs

We have explained how the graph itself will change due to the addition of macro-actions. In this section, we examine how the changes in the graph can provide intuitions about the changes to the basis functions and to the domain itself. We first start by examining the state graph. Figure 5.4 compares the invariant distribution of the two state graphs. The invariant distribution of the state graph of the MDP is displayed in Figure 5.4(a). The in-

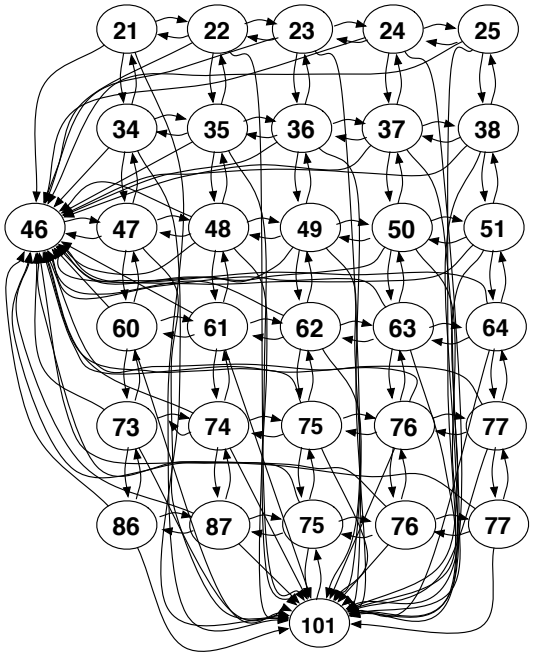


Figure 5.2. State graph for the upper right hand room showing transitions when the agent has access to both macro-actions and primitive actions.

variant distribution is fairly uniform with a slightly lower probability of being in the corner and doorway states. Figure 5.4(b) visualizes the invariant distribution of the state graph in the SMDP version of the four room gridworld. The addition of the hallway macro-actions causes the invariant distribution to be significantly higher in the hallway states and states that are close to these states. The states with the lowest values are those at the four corners. This result is expected and desirable. Options are often added to a domain to help an agent find key states that are important in solving a task. Thus, it is desirable that the invariant distribution of the random walk would be skewed towards states in the termination set of the options.

This change to the invariant distribution will also have an effect upon basis functions created using the directed graph Laplacian. Johns and Mahadevan (2007) discuss the effect of the the normalization by the Perron vector. We follow this analysis to understand the

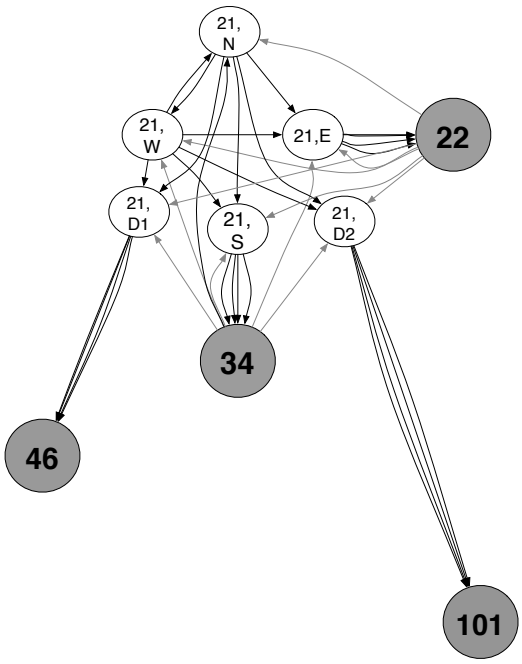
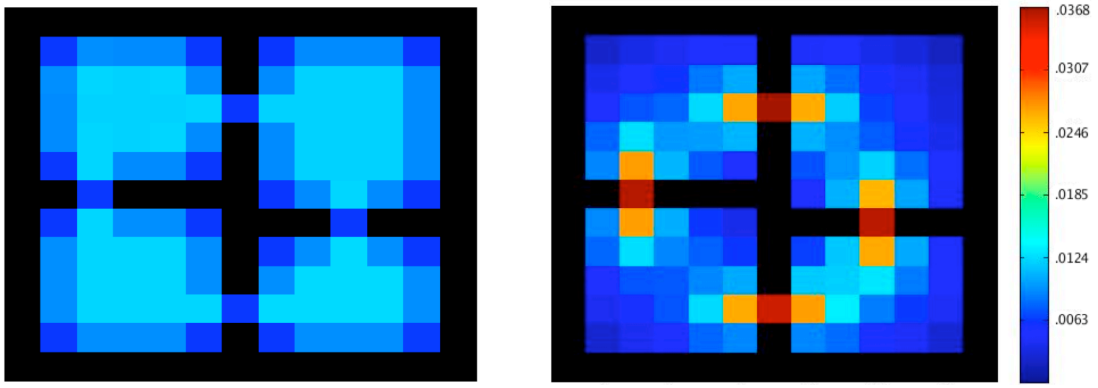


Figure 5.3. Transitions associated with nodes for the state-action pairs for state 1 when the doorway macro-actions are available.



(a) Invariant distribution when only primitive actions are available.

(b) Invariant distribution when options are available.

Figure 5.4. The invariant distribution of the four room gridworld with only primitive actions and with options.

effect of the addition of options upon the basis functions created from the directed graph Laplacian.

We have already mentioned that smoothness of a function f on a graph can be measured by the Sobolev norm and that functions that are smooth over the graph minimize this norm. The Sobolev norm for the directed graph Laplacian (Johns & Mahadevan, 2007) can be rewritten as:

$$\begin{aligned} ||f||^2_{\mathcal{H}^2} &= ||f||^2_2 + ||\nabla_f||^2_2 \\ &= \sum_{v \in V} |f(v)|^2 d_v + \sum_{u \to v} |f(u) - f(v)|^2 \frac{\psi_u}{d_u} W(u,v), \end{aligned}$$

where $u \to v$ indicates that there is an edge from vertex u to vertex v; ψ_u is the entry of vertex u in Ψ, the Perron vector, which is the invariant distribution upon convergence of a random walk on the graph. The second term of the Sobolev norm, $\langle f, \ L_d f \rangle = \sum_{u \to v} |f(u) - f(v)|^2 \frac{\psi_u}{d_u} W(u,v)$ is the smoothness constraint enforced by the directed graph Laplacian. Vertices with large values in Ψ will contribute more to the Sobolev norm. Thus, if a function f is smooth then $f(u) \approx f(v)$ when $u \to v$ and ψ_u is large compared to other vertices. As we have already discussed, the addition of options cause some states, specifically those in the termination set of the options, to have significantly higher values in the invariant distribution. Thus, for f to be smooth over the graph these states must be similar to vertices they connect to.

5.3 Learning Value Functions in Semi-Markov Decision Processes

In this section, we describe how Q-learning approaches have been extended to SMDPs. After experiencing a transition from state s to state s' under option o with duration N and experiencing reward r the following update is performed:

$$Q(s,o) \leftarrow Q(s,o) + \alpha[r + \gamma^N \max_{o' \in O(s')} Q(s',o') - Q(s,o)],$$

where r is the cumulative discounted reward over the option's duration.

We use SMDP Q(λ)-learning (Precup et al., 2000). We selected this method because it was off-policy learning (we wanted to be able to learn from samples experienced during our random walk) and because it learns more quickly than Q-learning. This method uses an ϵ-greedy policy for action selection and accumulating eligibility traces. Traces are set to zero when a random action is taken. The update rule for the parameters is given as:

$$\boldsymbol{\theta}_{t+K} = \boldsymbol{\theta}_t + \alpha\delta_t \boldsymbol{e}_t \tag{5.1}$$

where

$$\delta_t = r_s^o + \gamma^N \max_{o' \in O(s')|\boldsymbol{\theta}_t} \hat{Q}_t(s', o'|\boldsymbol{\theta}_t) - \hat{Q}_t(s, o|\boldsymbol{\theta}_t),$$

and

$$\boldsymbol{e}_t = \gamma^{N'} \lambda \boldsymbol{e}_{t-N'} + \nabla_{\boldsymbol{\theta}_t} \hat{Q}_t(s, o|\boldsymbol{\theta}_t),\ \boldsymbol{e}_0 = \mathbf{0}$$

and N' is the duration of the option selected immediately before transitioning to s. We use the following parameters($\gamma = .9, \epsilon = .1, \alpha = .01, \lambda = .9$) to learn a policy that approximately maximizes the agent's long-term reward.

We consider a learning problem in the four room gridworld domain. The agent may use both the primitive actions and options to learn to reach the goal. We first allow the agent to explore the environment selecting from primitive actions and available options randomly. We used 2000 episodes with 50 steps per episode. During the exploration period the agent's initial state is selected randomly. We perform this exploration only once. The agent then builds the graph from these samples and computes the basis functions. Figure 5.5 gives an algorithmic description of the process we use. During learning the agent starts in a random state and each episode is halted after 50 steps.

SA-RPI Algorithm $(\mathcal{D}, \gamma, \epsilon, k, \pi_0)$:
// $\mathcal{D}$: Source of samples (s, a, r, s')
// γ: Discount factor
// ϵ : Stopping criteria
// k: Number of basis functions
//π_0: Initial policy specified as a weight vector w_0

1. **Sample Collection:** Generate a set of samples $\mathcal{D}$, which consists of a state, action, reward, and nextstate, (s, a, r, s'). The samples are created using a series of exploratory trajectories using π_0 where π_0 selects the least frequently used action in any state.
2. **Representation Learning:**
   (a) Build a directed weighted graph G from $\mathcal{D}$ where V is the set of state-action pairs, E is the edge set, and W is the weight matrix using the on-policy approach in Figure 5.1.
   (b) Calculate the k lowest order eigenfunctions of the (combinatorial or normalized) graph Laplacian operator on G. These k eigenvectors make up the basis functions ϕ.
      i. Form the directed Laplacian per Equation 3.9 or 3.10.
      ii. Calculate ϕ by computing the eigenvectors of the directed Laplacian.
3. **Control Learning Phase**:
   Use Q(λ)-learning as the parameter estimation method to find the best policy π using the weight update in Equation 5.1.

Figure 5.5. RPI Framework for learning representation and control using state-action graphs in SMDPs.

We performed experiments to compare the two graph Laplacians on both state and state-action graphs. In these experiments we systematically varied the number of basis functions used in function approximation. In experiments using state graphs, we varied the number of basis functions from 24 to 120 in steps of 12 and from 144 to 1200 in steps of 24. In experiments using state-action graphs, we varied the number of basis functions from 3 to 10 in one step increments and from 20 to 600 in increments of 10. The results of each experiment was averaged over 200 trials and each experiment was performed for 300 episodes.

Figure 5.6 compares the number of steps taken by the agent to reach the goal when using the two types of graph Laplacians on both state and state-action graphs. The performance of the normalized and combinatorial graph Laplacians was similar on both the state and

state-option graphs, thus we plot only results using the normalized Laplacian. The best performance was in experiments using the state-option graphs with 260 basis functions (out of 616 possible basis functions). A similar number of basis functions created from the graph Laplacians of state graphs did not yield similar performance. Instead, performance using about 264 basis functions created from the state graph was similar to performing table lookup. Using more basis functions did not improve performance noticeably. All techniques using options outperformed experiments where the agent had access only to primitive actions.

Basis functions created from state-action graphs performed the best; we were not able to achieve similar performance with basis functions derived from the state graph. We also performed experiments varying α; however, the results were not significantly changed. Basis functions created from state-action graphs continually outperform basis functions created from state graphs.

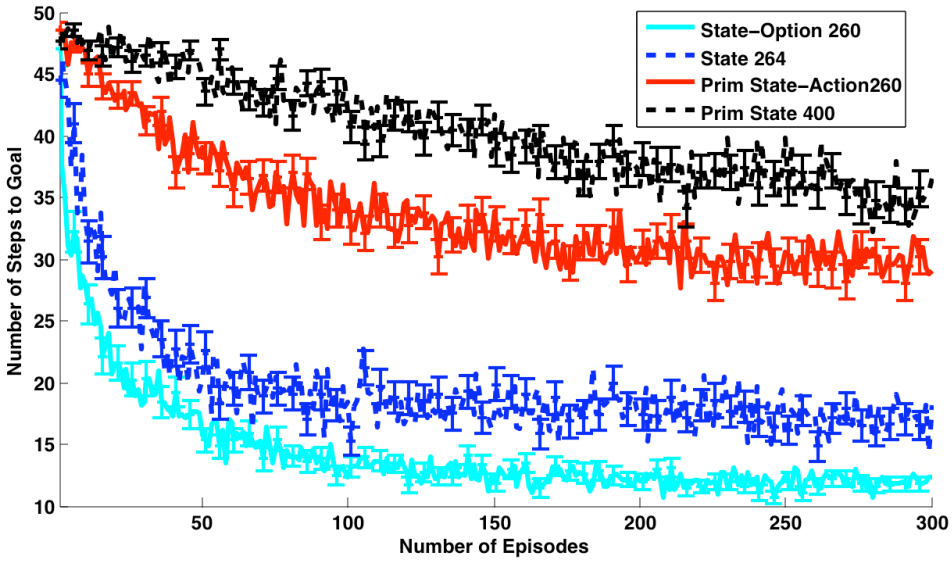


Figure 5.6. Steps to goal in the four room gridworld.

5.3.1 Eight Room Gridworld

We also ran experiments on an eight room gridworld shown in Figure 5.7 to demonstrate how copying can become increasingly expensive as the number of actions available to the agent increases. This domain consists of 325 states of which 210 are free states. In any free state the agent can perform one of four primitive actions: north, south, east or west. There is a 10% probability that an action will fail and the agent will remain in the same location. If the agent moves into a wall it remains in the same location. Rewards are zero on all state transitions except transitions into the goal state when the agent receives a reward of 100.

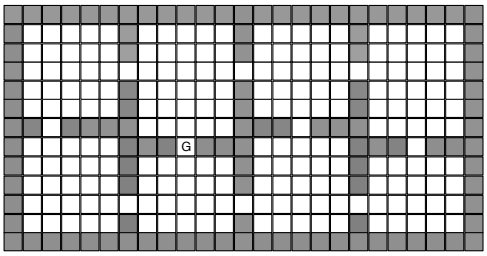


Figure 5.7. Eight room gridworld.

Hallway options are provided in each of four rooms. These options allow the agent to navigate from any location within one room to one of the hallway states that lead out of that room. Rooms adjacent to 3 doorways have 3 hallway options and rooms adjacent to 2 doorways have 2 hallway options. The initiation set, I, is comprised of all the states within the room. A hallway option's policy is optimal and cannot be terminated once selected until it reaches the goal state. Hallway states do not have hallway options available to them.

The learning problem in this domain is that the agent must use the 20 multi-step hallway options and primitive actions to learn to reach the goal. In this domain, the agent's initial state is always a random state. We first allow the agent to explore the environment selecting from primitive actions and available options randomly. We used 4000 episodes with 50 steps per episode. We perform this exploration only once. The agent then builds the graph from these samples and computes the basis functions. We use SMDP Q(λ)-learning($\gamma =$

$.9, \epsilon = .1, \alpha = .01$). The agent is allowed 100 steps per learning episode and the Q-function is initialized to zero.

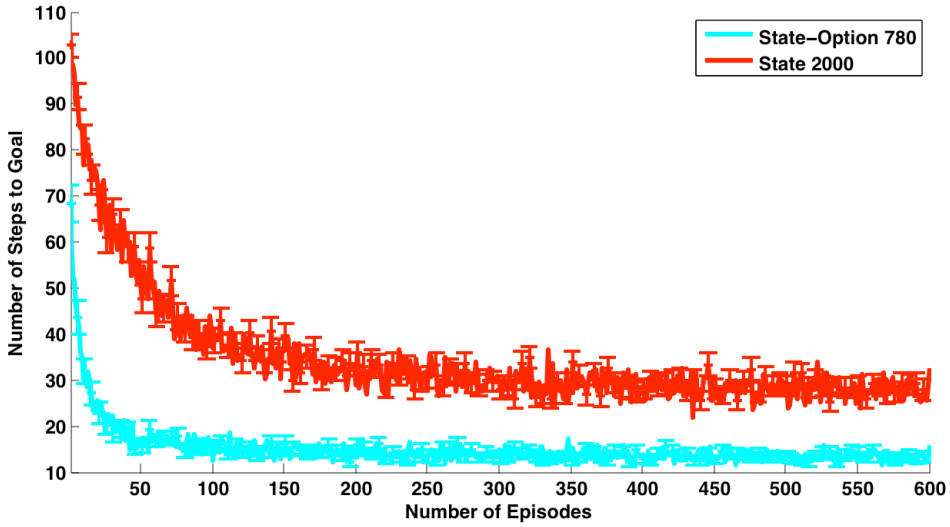


Figure 5.8. Steps to goal in the eight room gridworld.

We performed similar experiments to compare the two graph Laplacians on both state and state-action graphs in the eight room gridworld. In these experiments, we systematically varied the number of basis functions used in function approximation. The results of each experiment was averaged over 200 trials and each experiment was performed for 600 episodes. Figure 5.8 shows that once again the state-option graphs out perform the state graphs. In the eight room grid world we use 780 (out of 2520) basis functions on the state-option graph. The best results using a state graph required 2400 basis functions. Both results are shown using the normalized graph Laplacian.

5.3.2 Comparison of Graph Creation Techniques

Earlier in the chapter we proposed a set of weightings for state-action graphs. In this section, we will discuss other techniques for weighting the graph and experimentally evaluate these approaches. In this experiment, we examine how weightings affect the ability

to learn. We examine weightings that take into account the average duration of the option, weightings that use information about the likelihood of the transition and weightings that combine these two measures as well as a baseline 0 or 1 weighting. Table 5.2 shows the nine weightings we compared.

Weighting 1	$W(i,j) = \frac{1}{avetime(i,j)} \frac{count(i,j)}{count(i)}$
Weighting 2	$W(i,j) = 1$ if edge else 0
Weighting 3	$W(i,j) = \frac{1}{avetime(i,j)}$
Weighting 4	$W(i,j) = avetime(i,j)$
Weighting 5	$W(i,j) = \frac{count(i,j)}{count(i)}$
Weighting 6	$W(i,j) = (\sum_{t=1}^{time(i,j)} \gamma^t)^{-1}$
Weighting 7	$W(i,j) = e^{-time(i,j)}$
Weighting 8	$W(i,j) = (\sum_{t=1}^{time(i,j)} \gamma^t)^{-1} \frac{count(i,j)}{count(i)}$
Weighting 9	$W(i,j) = e^{-time(i,j)} \frac{count(i,j)}{count(i)}$

Table 5.2. Weightings used in comparison experiments.

Figure 5.9 displays an overview of the results from our weighting experiment. All results used 260 basis functions and the only difference between the experiments is the weightings for W. Our results demonstrate that using both information about time and likelihood in the weight is the most successful approach. Weighting 7 and Weighting 9 are not shown on the figure. However, experiments for Weighting 7 had similar performance to Weighting 6 and the results for Weighting 9 were similar to Weighting 1. These results indicate that adding in both time and likelihood information gave the best performance.

5.4 Conclusion

In this chapter, we extended our approach for automatically constructing basis functions for action-value functions to SMDPs. We describe how both state and state-action graphs can be modified to incorporate information about options.

We experimentally evaluated the performance of these basis functions for learning action-value functions. Our results demonstrate that basis functions created from the state-

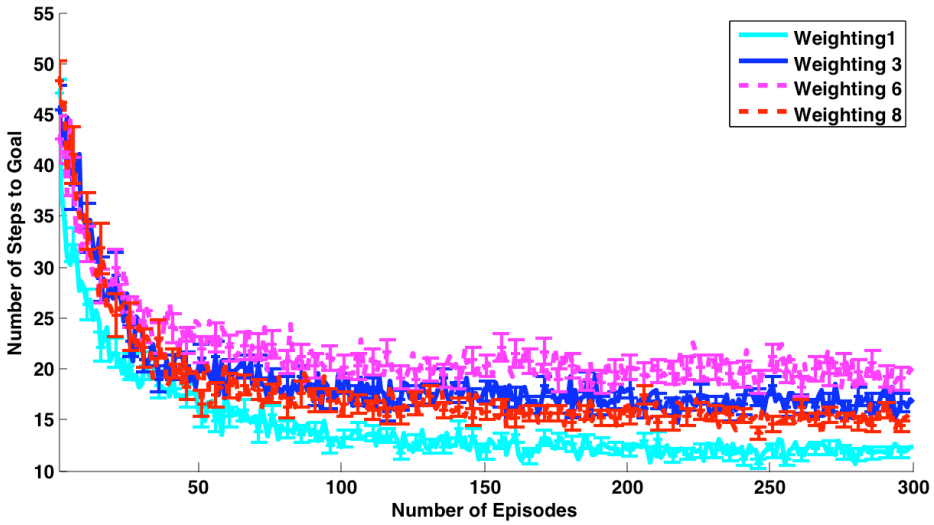


Figure 5.9. Weighting comparison

action graph significantly improve learning performance when compared to basis functions created on the state-graph. Additionally, our results for state-action graphs show that the best weightings for these graphs include temporal and likelihood information.

CHAPTER 6

REPRESENTATION DISCOVERY FOR HIERARCHICAL REINFORCEMENT LEARNING

Chapter 5 introduced a technique for representation discovery when the agent has access to macro actions, each with a fixed policy. In this chapter, we generalize basis function construction to multi-level hierarchies where the agent learns at multiple levels of temporal abstraction simultaneously. Multi-level task hierarchies allow problems to be decomposed into smaller subproblems. There are several advantages to decomposing the problem into smaller subtasks. First, policies learned in a subtask can be reused for multiple parent tasks. Second, value functions for a subtask can also be shared, which significantly decreases the time required to learn the value function of a new parent task. Third, task hierarchies create opportunities for state abstraction, which allows the value function to be represented compactly. These features all help speed up the learning process. In this chapter, we introduce an approach to basis function construction for problems where the agent has access to a task hierarchy. Basis functions can substantially speed up learning since they provide generalizations, and task hierarchies can help automatic basis function construction approaches scale to larger problems.

Definition 6.1 Automatic Basis Construction Problem for Multi-Level Task Hierarchies: *Given a Markov Decision Process $M = (S, A, P, R)$ and a task hierarchy H, automatically construct a low-dimensional representation Φ. The construction method should leverage H to create a compact representation Φ that respects the hierarchy. Φ should be constructed such that the solution to M calculated using Φ closely approximates the solution of the original MDP M.*

Task Hierarchy for Taxi

Before continuing further, we describe the taxi task, which we will use as an example throughout this chapter. The taxi task domain (Dietterich, 1998) is pictured in Figure 6.1. The taxi task is defined as a grid of 25 states with four colored locations, red (R), green (G), yellow (Y), and blue (B). The task is for the agent, the taxi, to pick up the passenger located on one of the colored locations and drop the passenger at the desired destination. The state can be written as a vector of variables. Each state contains the location of the taxi, the passenger location, and the passenger destination. There are 6 primitive actions in this domain. Four of these actions are navigation actions: *north*, *east*, *south*, and *west*. The other two actions access the passenger location, *pickup* and *putdown*. Each action receives a reward of -1. If the passenger is *putdown* at the intended destination, a reward of $+20$ is given. If the taxi attempts to *pickup* a nonexistent passenger or *putdown* the passenger at the wrong destination, a reward of -10 is received. If the taxi runs into the wall, it remains in the same state and receives a reward of -1.

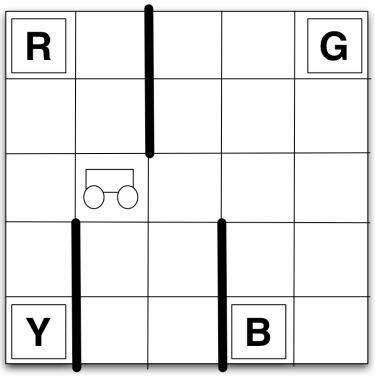


Figure 6.1. Taxi Domain

The task hierarchy, pictured in Figure 6.2, is defined as follows. The root node is defined over all states and state variables and can select one of two subtasks, *get* and *put*. The *get* action can only be selected when the passenger is not located in the taxi and the *put* action can only be selected when the passenger is located in the taxi. No learning occurs

at the root subtask because each state has only one action available to it at any given time. The *get* action only considers the taxi location and the passenger location. It has access to two actions, *navigate(p)*, and *pickup*. The *put* action considers only the taxi location and the passenger destination. It has access to two actions: *navigate(p)*, and *putdown*. The *navigate* action takes 4 parameters that indicate which of the 4 locations it can navigate to and has access to the 4 navigation actions.

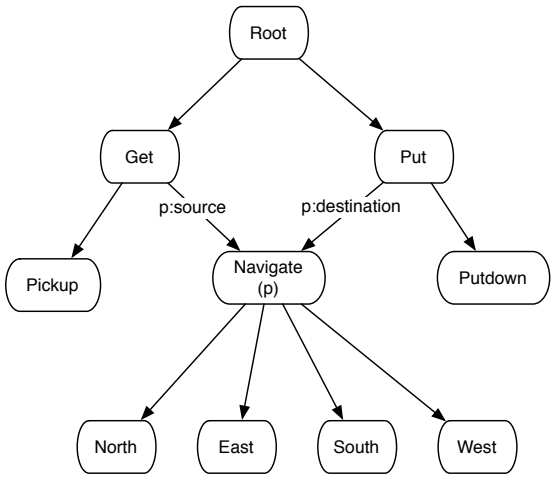


Figure 6.2. Hierarchy for the Taxi Domain

Considerations for Automatic Basis Function Construction Techniques

Task hierarchies provide opportunities to speed up learning through policy reuse, value function reuse, and state abstractions. Function approximation techniques combined with task hierarchies provide a powerful opportunity to create compact representations through generalization. We describe some considerations that arise when constructing basis functions for multi-level task hierarchies.

The first consideration is whether information about the reward function should be incorporated during basis function construction. Research on basis function construction has largely been divided into two categories: reward sensitive approaches (Keller et al., 2006; Parr et al., 2007; Petrik, 2007) and reward insensitive approaches (Mahadevan, 2005;

Sugiyama et al., 2007). Reward insensitive basis functions are an appropriate choice for low level subtasks that are often parameterized because only one set of basis functions must be built, rather than a set for each parameterization. For example, in the taxi task the navigate subtask has four parameters; reward sensitive approaches would create four sets of basis functions for this subtask that correspond to the different reward functions for each parameterization.

The second consideration is that temporal locality and spatial locality may no longer be correlated in hierarchical reinforcement learning (HRL) tasks. For some levels of the hierarchy, states that are sequential in the agent's decision making may no longer be close in terms of spatial locality. Figure 6.3 shows an instance of the taxi *get* task where the passenger is located on the green square. If the taxi is located at the darkly shaded state, it can travel to one of the four colored states. Therefore in the state graph, this state will be connected to the four colored states by the *navigate*(p) action. In the *get* task the agent will not observe a transition from the darkly shaded state to the lightly shaded state. Thus the states are not connected, even though these states have similar values for each of the navigate actions.

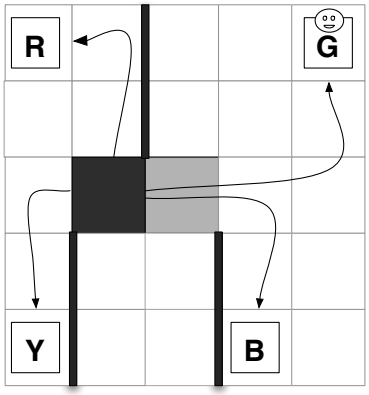
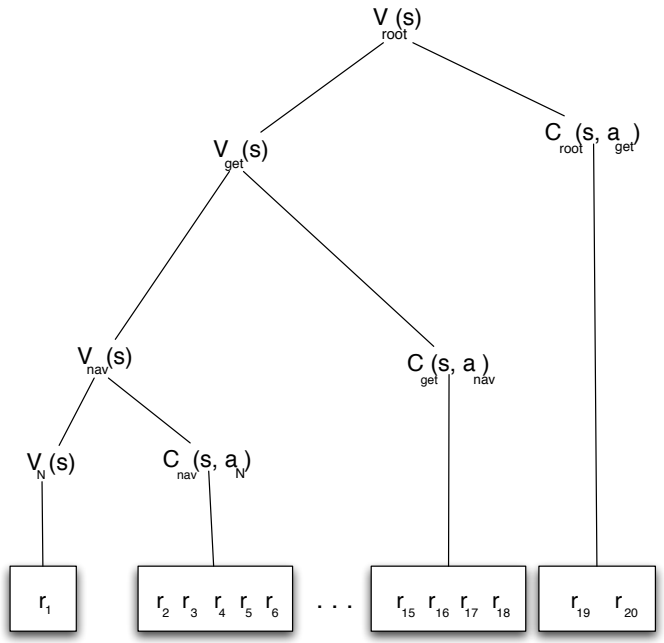


Figure 6.3. An example of the taxi *get* task where the taxi must pick up the passenger located in the green square.

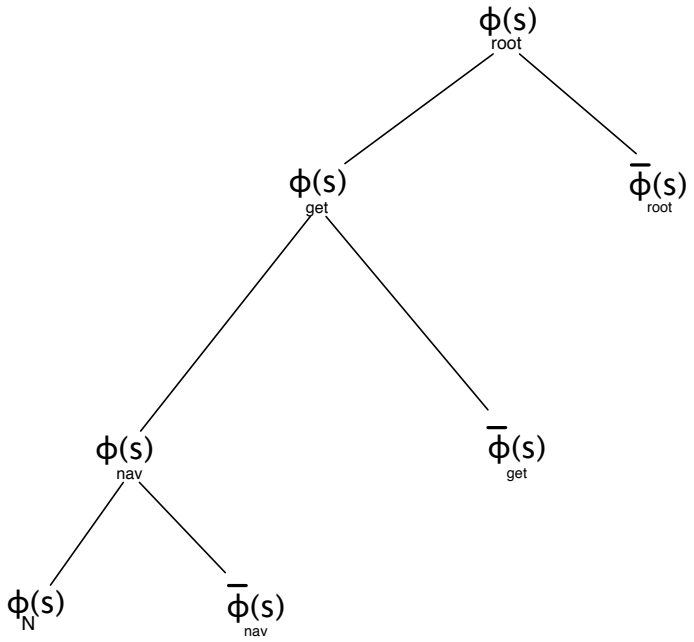
The third consideration is that task hierarchies are constructed to decompose problems into simple subproblems. These subproblems allow both the policies and value functions of subtasks to be shared. Figure 6.4(a) shows how the Q-value function decomposes under the MAXQ framework (Dietterich, 2000) into two parts: $V_a(s)$ the expected sum of rewards obtained while executing action a and the completion function $C_i(s, a)$, the expected cumulative reward for subtask i following the current policy π_i after action a is taken in state s. Figure 6.4(a) specifically visualizes the Q-function decomposition for the taxi task.

In order to scale, the representations created for HRL problems should decompose recursively in a similar manner. Lower level representations could be reused when constructing basis functions at a higher level. Figure 6.4(b) is a visualization of how basis functions might decompose according to the hierarchy. For a subtask i, the basis functions for state s can be decomposed into two parts: $\bar{\phi}_i(s)$ the "local" basis functions constructed at subtask i and $\phi_a(s)$ the basis functions from child subtasks, where a is one of the child subtasks. Figure 6.4(b) specifically illustrates how the basis functions decompose for the taxi task. In this work, $\bar{\phi}_i(s)$ is automatically constructed using spectral analysis of a graph G_i that is built for subtask i from the agent's experience. However, other automatic basis function construction approaches, such as BEBFs (Parr et al., 2007), could be used to construct the basis functions.

Automatic basis function construction approaches must address these considerations. In this chapter, we describe an approach to representation discovery for HRL using eigenvectors of the graph Laplacian. Our approach leverages the hierarchy by constructing basis functions over the abstract state space introduced by the task hierarchy. It also constructs basis functions that decompose according to the hierarchy. We evaluate our approach experimentally using the MAXQ learning framework (Dietterich, 1998).



(a) Value function decomposition due to the task hierarchy for the taxi task.



(b) Representation decomposition based on the task hierarchy for the taxi task.

Figure 6.4. We explore an approach to basis function construction that exploits the value function decomposition defined by a fixed task hierarchy.

6.1 Hierarchical Reinforcement Learning

Hierarchical reinforcement learning algorithms constrain policies via a hierarchy (Barto & Mahadevan, 2003). These algorithms allow the agent to select actions that take more than one time step. Often hierarchical RL algorithms use semi-Markov decision processes (SMDPs) as a model. SMDPs are a generalization of MDPs in which actions are no longer assumed to take a single time step and may have varied durations. An SMDP is defined as a tuple $M = (S, A, P, R)$. All components have the same definition as in an MDP, except the transition probability function P and the reward function R. S is the set of states, and A is the set of actions the agent may take at each decision point. The transition probability function P is modified to take into account the duration of the actions. P is now a multi-step transition probability function, where $P(s', N|s, a)$ denotes the probability that action a taken in state s will cause a transition to state s' in N time steps. The reward function is also modified to take into account the duration of the actions. Rewards can accumulate over the entire duration of an action. The reward function $R(s', N|s, a)$ is the expected reward received from selecting action a in state s and transitioning to state s' with a duration of N time steps. An SMDP can be seen as representing the system at decision points, while an MDP represents the system at all times.

In this chapter, we are specifically interested in frameworks where the temporally extended actions are not assumed to have a fixed policy and the agent learns at multiple levels of abstraction simultaneously. In HRL tasks, agents solve the SMDP by learning the function $Q(s, a)$, which is the expected sum of discounted reward for taking action a in state s. It is important to note that in this context a may be either a temporally extended action or a primitive action.

6.1.1 Task Hierarchies for Reinforcement Learning

In this section, we formalize multi-level task hierarchies for RL. A task hierarchy decomposes an MDP M into a set of subtasks $\{M_0, M_1, ..., M_n\}$, which can be modeled

as SMDPs. M_0 is the root subtask that solves M. A subtask is defined to be a tuple $M_i = (\beta_i, A_i, \tilde{R}_i)$.

- β_i is the termination predicate that partitions S into a set of active states S_i and a set of terminal states β_i. The policy π_i for subtask M_i can only be executed if the current state s is in S_i. If the M_i ever enters a state in β_i while executing, then M_i terminates.

- A_i is a set of actions that can be performed to achieve subtask M_i. Each action can either be a primitive actions from A or another subtask. If a subtask is called from M_i, it is called the child of subtask i. No subtask can call itself either directly or indirectly.

- $\tilde{R}_i(s)$ is the deterministic pseudo-reward function. It is a reward function specific to M_i. R_i is defined for all states $s \in \beta_i$ and tells how desirable a state is for the subtask. The pseudo-reward is only used during learning.

Task hierarchies may also have parameterized subtasks. If M_j is a parameterized subtask, it is as if this task occurs many times in A_i, where M_i is the parent task. Each parameter of M_j specifies a distinct task. β_i and $\tilde{R}_i$ are redefined as $\beta_i(s, p)$ and $\tilde{R}_i(s', p)$, where p is the parameter's value. If a subtask's parameter has many values, it is the same as creating a large number of subtasks, which must all be learned. It also creates a large number of possible actions for a parent task.

A hierarchical policy $\pi = \{\pi_0, \ldots, \pi_m\}$ is a set containing a policy for each subtask in the task hierarchy. In each subtask, π_i takes a state and returns a primitive action or subtask to be executed. $P_i^{\pi}(s', N|s, a)$ is the probability transition function for a hierarchical policy at level i, where $s, s' \in S_i$ and $a \in A_i$.

6.1.2 State Abstraction for Multi-level Hierarchies

One of the most significant advantages of HRL is that task hierarchies allow state abstractions to occur through an abstraction function χ. Each state s can be written as a vector

of variables X. X_i is the subset of state variables that are relevant to subtask i. $X_{i,j}$ is the jth variable for subtask i. A state $\boldsymbol{x}_i$ defines a value $x_{i,j} \in Dom(X_{i,j})$ for each variable $X_{i,j}$. χ_i is a function that maps a state s onto only the variables in X_i. Since subtasks can ignore certain portions of the state space, the number of distinct values required to represent the value function can be significantly smaller. This significantly speeds up learning.

Dietterich (2000) discusses three types of abstraction within HRL tasks. The first type of abstraction eliminates irrelevant variables within a subtask. Subtasks higher in the task hierarchy tend to have more relevant variables, while subtasks lower in the task hierarchy tend to have fewer relevant variables. The second type of abstraction results from the structure of the task hierarchy itself. Large parts of a subtask's state space may not be reachable due to the termination conditions of its ancestors in the task hierarchy. The third type of abstraction involves *funnel* actions. Funnel actions are macro-actions that move the agent from some large number of potential initial states to a small number of resulting states. The abstractions Dietterich (2000) describes for funnel actions are more specific to the completion function C, and may not be general enough to be included for hierarchical Q-learning. They also do not hold true when the agent is maximizing discounted reward. Dietterich (2000) assumed that χ is constructed by a human designer. Some research, such as Ravindran and Barto (2003), examines automatically building abstractions. Ravindran and Barto (2003) use homomorphisms to create abstractions when the agent has access to temporally extended actions.

Definition 6.2 State-abstracted task hierarchy: *Assume each state s can be written as the values of a vector of state variables. Given an MDP M and a task hierarchy H, the state variables for each subtask i can be partitioned into two sets X_i and Y_i, where Y_i is the set of state variables irrelevant to the task. χ_i projects s onto only the values of the variables in X_i. When combined with χ, H is called a state-abstracted task hierarchy.*

A state-abstracted task hierarchy reduces the size of the learning problem because an abstract hierarchical policy can be defined over the reduced space.

Definition 6.3 Abstract Hierarchical Policy: *For MDP M with a state-abstracted task hierarchy H and χ, an abstract hierarchical policy is a hierarchical policy in which each subtask i has a policy π_i that satisfies the following condition: for any two states s_1 and s_2 such that $\chi_i(s_1) = \chi_i(s_2)$ then $\pi_i(s_1) = \pi_i(s_2)$. When π_i is a stochastic policy, such as during exploration, the probability distribution for action selection in s_1 and s_2 will be the same.*

6.1.3 Solving HRL tasks

Each subtask M_i has a value function $Q_i(s, a)$ that defines the value of taking an action a in state s according to the real reward function R. $Q_i(s, a)$ is used to derive a policy π_i, typically by selecting the action with the maximum Q value for s.

In the MAXQ framework (Dietterich, 1998) the value function is decomposed based upon the hierarchy. MAXQ defines Q_i recursively as:

$$Q_i(s, a) = V_a(s) + C_i(s, a)$$

where

$$V_i(s) = \begin{cases} \max_a Q_i(s, a) & \text{if } i \text{ is composite} \\ V_i(s) & \text{if } i \text{ is primitive.} \end{cases}$$

$V_a(s)$ is the expected sum of rewards obtained while executing action a. The completion function $C_i(s, a)$ is the expected discounted cumulative reward for subtask i following the current policy π_i after action a is taken in state s.

$\tilde{C}$ is the completion function that incorporates both $\tilde{R}_i$ and R, and is used only inside the subtask to calculate the optimal policy of subtask i. $\tilde{Q}_i$ is defined as $\tilde{Q}_i(s, a) = V_a(s) + \tilde{C}_i(s, a)$. $\tilde{Q}$ is used to select the action. If $\tilde{R}_i$ is zero, then C and $\tilde{C}$ will be identical.

6.1.3.1 Function Approximation for HRL

When performing function approximation in HRL, each subtask has a set of basis functions Φ_i and a set of k weights $\boldsymbol{\theta}_i$ that are used to calculate the value function Q. $\phi_i(s,a)$ is a k length feature vector for state s and action a.

In MAXQ, the completion function for subtask i at time t $C_{i,t}(s,a)$ is approximated by $\hat{C}_{i,t}(s,a) = \sum_{j=1}^{k} \phi_{i,j}(s,a)\boldsymbol{\theta}_{i,j,t}$. The update rule for the parameters is given as:

$$\boldsymbol{\theta}_{i,(t+N)} = \boldsymbol{\theta}_{i,t} + \alpha_i[\gamma^N(\max_{a' \in A(s')} \hat{C}_{i,t}(s',a'|\boldsymbol{\theta}_{i,t}) + V_{a',t}(s')) - \hat{C}_{i,t}(s,a|\boldsymbol{\theta}_{i,t})] \cdot \phi_i(s,a).$$

In our experiments we use Q(λ) learning with replacing traces. The update rules for this are:

$$\boldsymbol{\theta}_{i,(t+N)} = \boldsymbol{\theta}_{i,t} + \alpha\delta_{i,t}\boldsymbol{e}_{i,t}, \text{ where}$$
$$\boldsymbol{e}_{i,t} = \gamma^N\lambda\boldsymbol{e}_{i,t-N} + \phi_i(s,a), \boldsymbol{e}_0 = \mathbf{0}$$

and

$$\delta_{i,t} = \gamma^N(\max_{a' \in A(s')} \hat{C}_{i,t}(s',a'|\boldsymbol{\theta}_{i,t}) + V_{a',t}(s')) - \hat{C}_{i,t}(s,a|\boldsymbol{\theta}_{i,t}) \tag{6.1}$$
$$\tilde{\delta}_{i,t} = \gamma^N(\tilde{R}(s,a) + \max_{a' \in A(s')} \hat{\tilde{C}}_{i,t}(s',a'|\tilde{\boldsymbol{\theta}}_{i,t}) + V_{a',t}(s')) - \hat{\tilde{C}}_{i,t}(s,a|\tilde{\boldsymbol{\theta}}_{i,t}).$$

6.2 Automatic Basis Function Construction for Multi-level Hierarchies

While the state abstractions that are often provided with task hierarchies can be extremely effective at creating compact representations, function approximation can still provide powerful opportunities for generalization. In this section, we describe an approach for automatic basis function construction for multilevel task hierarchies.

We focus on the graph Laplacian approach to automatic basis function construction (Mahadevan, 2005). In this approach, the agent automatically constructs basis functions

by first exploring the environment and collecting a set of samples. These samples are then used to create a graph where the vertices are states and edges are actions. Basis functions are created by calculating the eigenvectors of the Laplacian of the graph. We describe how we adapted this approach for multi-level task hierarchies.

One of the reasons HRL is useful is that value functions have been shown to decompose with the hierarchy, as long as the hierarchy is well constructed. The intuition behind our approach to representation discovery for HRL problems is that basis function construction should decompose in a similar way. The first step in our approach constructs a graph for each subtask from the agent's experience. We discuss how graph construction can leverage the abstractions provided with the task hierarchy. We also introduce an approach to creating abstractions based upon the graph structure. The second step in our approach constructs basis functions recursively from child subtasks. Figure 6.5 shows our approach to HRL with representation discovery.

6.2.1 Graph Creation for Multi-level Task Hierarchies

The first step to our approach for representation discovery for multi-level task hierarchies is to perform sample collection, such that each subtask i has a set of samples $\mathcal{D}_i$. Each sample in $\mathcal{D}_i$ consists of a state, action, reward, and next state, (s, a, r, s'). The agent constructs a graph from $\mathcal{D}_i$. The agent can leverage a state-abstracted task hierarchy by building the graph in the abstract space defined by χ_i. The graph is built such that $\chi_i(s_1)$ is connected to $\chi_i(s_2)$, if the agent experienced a transition from $\chi_i(s_1)$ to $\chi_i(s_2)$ in $\mathcal{D}_i$. We call a graph constructed over the abstract state space a state-abstracted graph. Figure 6.6 describes how a graph can be created; this approach is similar to the approach in Chapter 5 but uses the abstraction function χ.

Definition 6.4 State-abstracted Graph: *For an MDP M with a state-abstracted task hierarchy, a state-abstracted graph G_i can be constructed for subtask i over the abstract state space defined by χ_i. The vertices V correspond to the set of abstract states $\chi_i(S)$ or a*

`HRL-RD Algorithm` (Subtask i, State s, Initial Samples $\mathcal{D}$, Number of basis functions k_i, Initial Policy $\pi_{0,i}, \gamma, \lambda, \epsilon$):

if i is a primitive subtask
  execute i, receive r, and
    observe the next state s'
  $V_{t+1,i}(s) := (1 - \alpha_t(i)) \cdot V_t(i, s) + \alpha_i(t) \cdot r_t$
  return $s', 1$
else
  if first time executing i call `CreateBasis`$(i, \mathcal{D}, k_i, \pi_{0,i})$ found in Figure 6.6.
  $\boldsymbol{e} = \mathbf{0}, \bar{N} = 0$
  while $\beta_i(s)$ is false **do**
    $a^* = \text{argmax}_{a'}[\tilde{Q}_{i,t}(s', a'|\boldsymbol{\theta}_{i,t})]$
    choose an action a according to the current policy π_i
    if $a = a^*$
      $\boldsymbol{e}_i = \gamma^N \lambda \boldsymbol{e}_i$
    else
      $\boldsymbol{e}_i = \mathbf{0}$
    end
    $\boldsymbol{e}_i = \boldsymbol{e}_i + \phi_i(s, a)$
    $(s', \tau) =$ `HRL-RD` (a, s)
    Use update rules from Equation 6.1
    $\boldsymbol{\theta}_{i,(t+N)} = \boldsymbol{\theta}_{i,t} + \alpha_i \delta_{i,t} \boldsymbol{e}_i$
    $\tilde{\boldsymbol{\theta}}_{i,(t+N)} = \tilde{\boldsymbol{\theta}}_{i,t} + \alpha_i \tilde{\delta}_{i,t} \boldsymbol{e}_i$
    $s = s'$
    $\bar{N} = \bar{N} + N$
    return $s', \bar{N}$
    end // while
  end // else

Figure 6.5. HRL Algorithm with representation discovery.

subset of the abstract states. An edge exists between v_1 and v_2 if there is an action $a \in A_i$ that causes a transition between the corresponding abstract states.

6.2.1.1 State-abstracted graph for the *Get* Task

We use the get task to show an example of a state-abstracted graph. Figure 6.7 shows the state-abstracted graph for the *get* task. $\chi_{get}(s)$ maps each state s to an abstract state

`CreateBasis Algorithm`(Subtask i, Samples $\mathcal{D}$, Number of local basis functions k_i, Initial policy π_0)

1. **Sample Collection:**
   (a) **Exploration:** Generate a set of samples $\mathcal{D}_i$, which consists of a state, action, reward, and nextstate, (s, a, r, s', N) for subtask, i according to π_0. N is the number of steps a took to complete.
   (b) **Subsampling Step (optional):** Form a subset of samples $\mathcal{D}_i \in \mathcal{D}$ by some subsampling method.
2. **Representation Learning:**
   (a) **if** GraphReduction will be performed
   Build an edge labeled graph $\mathcal{G}_i = (\mathcal{V}, \mathcal{E}, \mathcal{Z}, \mathcal{W})$ from $\mathcal{D}_i$ where $\mathcal{Z}$ are labels over the edge set $\mathcal{E}$. State v_1 is connected to state v_2 if $\chi(s_1)$ and $\chi(s_2)$ are linked temporally in $\mathcal{D}_i$ by an action a.
   $\mathcal{Z}(\chi_i(s_1), \chi_i(s_2)) = a$.
   (b) **else** Build an graph $\mathcal{G}_i = (\mathcal{V}, \mathcal{E}, \mathcal{W})$ from $\mathcal{D}_i$ where state v_1 is connected to state v_2 if $\chi(s_1)$ and $\chi(s_2)$ are linked temporally in $\mathcal{D}_i$.
   (c) G_i=`GraphReduction` $(\mathcal{G}_i, k_i)$ as found in Figure 6.8.
   (d) Calculate the k_i lowest order eigenfunctions of the graph Laplacian of G_i.

Figure 6.6. CreateBasis Algorithm for Hierarchical Reinforcement Learning.

$\boldsymbol{x}_{get} \in X_{get}$ where $X_{get} = \{$passenger position, taxi position$\}$. Each vertex in Figure 6.7 is an abstract state $\boldsymbol{x}_{get}$.

The four clusters of vertices correspond to a clustering of the states according to their values for the passenger location. Since there are four different values for the passenger location there are four cluster. Within each cluster, the darker vertices correspond to states where the taxi is located on one of the colored grid states. The light vertices in the graph are not connected to one another but only to the dark colored vertices. This is because the *get* subtask can only execute the *navigate* and *pickup* actions, and the *navigate* action leads to one of the four colored grid states. The light edges refer to edges caused by the *navigate* subtask. Dark edges refer to edges caused by primitive actions, in this case the *pickup* action. Each cluster has only one vertex with a dark edge. This vertex represents

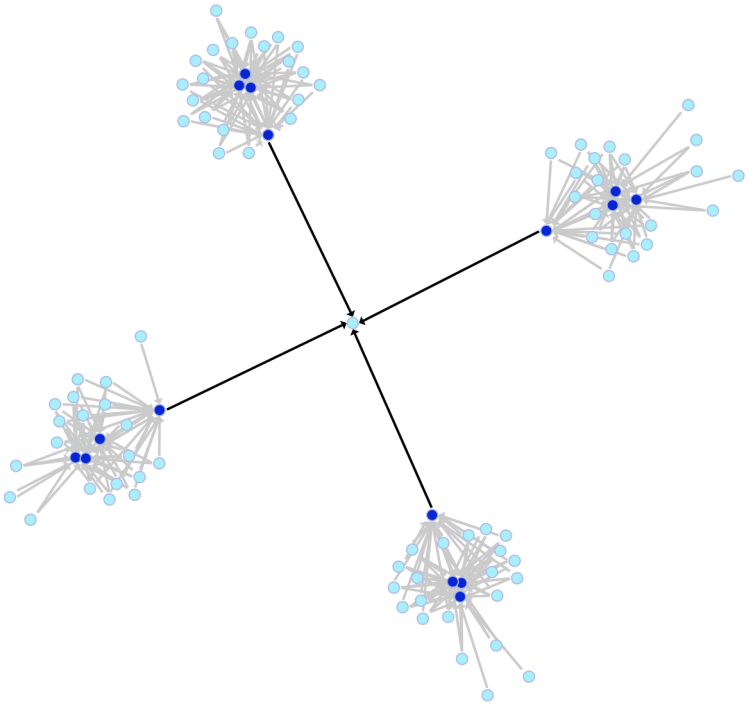


Figure 6.7. State-abstracted graph of the *get* subtask.

the state where the taxi is located in the same state as the passenger location. The center vertex represents the terminal state where the passenger is no longer in the taxi.

6.2.1.2 Building a Reduced Graph

In this section, we describe how state abstractions can be created using a graph reduction algorithm. The approach uses only properties of the graph to construct the abstraction. Our approach to graph reduction requires that the original graph $\mathcal{G}_i$ be an edge labeled graph. We define an edge labeled graph to be $G = (V, E, Z, W)$, where V is the set of vertices, E is the edge set, Z is a set of labels over E, and W is the weight matrix. $\mathcal{G}_i$ must be constructed such that the Z is the action a that caused the transition between v_1 and v_2. $\mathcal{G}_i$ may be a state or state-abstracted graph.

Definition 6.5 Reduced Graph: *A reduced graph can be constructed for subtask i from a graph $\mathcal{G}_i$. Two vertices v_1 and v_2 correspond to states, or abstract states, s_1 and s_2. v_1*

and v_2 can be represented as the same abstract vertex $\tilde{v}$, if the state variables for M_i can be divided into two groups X_i and Y_i such that:

- *s_1 and s_2 differ only in their values of Y_i*
- *v_1 and v_2 are connected to the same set of vertices in the graph and the labels $z \in Z$ for the respective edges are the same.*

v_1 and v_2 are merged into an abstract vertex $\tilde{v}$ corresponding to the subset of state variables X_i.

The graph reduction algorithm creates a reduced graph if M does not have an abstraction function χ associated with H or if χ exists but the state-abstracted graph G_i can be further compressed. If no nodes are merged, the graph will be the original graph. Figure 6.8 contains the algorithm used to transform the state graph into the reduced graph and create basis functions from the reduced graph.

`GraphReduction Algorithm`(Original Graph $\mathcal{G}_o, k_i$)
Create reduced graph, $G_i = (V, E, W)$, from $\mathcal{G}_o$
$V = \mathcal{V}_o$

For all $v_1 \in V$
  Loop through $v_2 \in V$
    V_1 is the set of vertices such that $v' \in V_1 \implies v_1 \rightarrow v'$
    V_2 is the set of vertices such that $v' \in V_2 \implies v_2 \rightarrow v'$
    If $V_1 = V_2$ **and** the labels over the edges are the same **and** $s_1 = (\mathbf{x}_i, \mathbf{y}_1)$ and $s_2 = (\mathbf{x}_i, \mathbf{y}_2)$
      Then merge v_1 and v_2 into an abstract node $\tilde{v}$ corresponding to the state variables X_i

Return G_i

Figure 6.8. Graph Reduction Algorithm

6.2.1.3 Reduced graph for the *Get* Task

The reduced graph for the *get* task is shown in Figure 6.9. The outer four nodes are abstract nodes corresponding to states where the taxi is not in one of the colored grid locations. The four inner states correspond to the bottleneck states when the agent is in

the same location as the passenger. The center state represents when the passenger has been picked up and is in the taxi. Basis functions for the *get* task will be constructed using eigenvectors of the reduced graph and basis functions from the navigate child subtask.

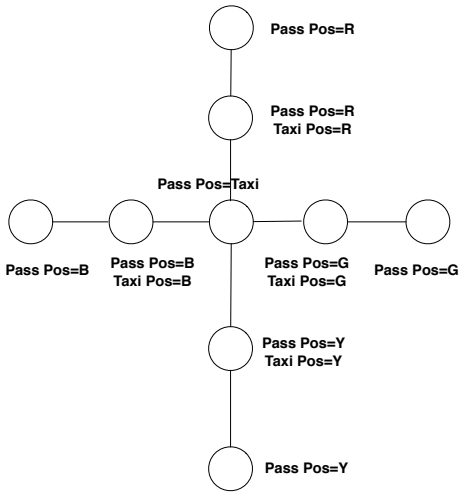


Figure 6.9. Reduced graph for the *get* task.

The reduced graph in Figure 6.9 is just one of the reduced graphs that will be constructed for the taxi task. Figure 6.10 shows all of the reduced graphs for the taxi task.

6.2.1.4 Generating Hierarchical Basis Functions

The basis functions for a subtask i are automatically constructed by first generating the *local basis functions* $\bar{\Phi}_i$. $\bar{\Phi}_i$ is constructed from the eigenvectors of the graph Laplacian of G_i, as described in Chapter 3. These basis functions are concatenated with basis functions recursively gathered from the child subtasks. This means that the basis functions are no longer guaranteed to be linearly independent. If necessary, the bases can be reorthogonalized using Gram-Schmidt or QR decomposition.

We define φ to be a compression of the state space. Compressions can be abstractions defined by χ, such as those proposed by Dietterich (2000) as well as those from the reduced graph. Compressions can also be those constructed through spectral graph analysis. For

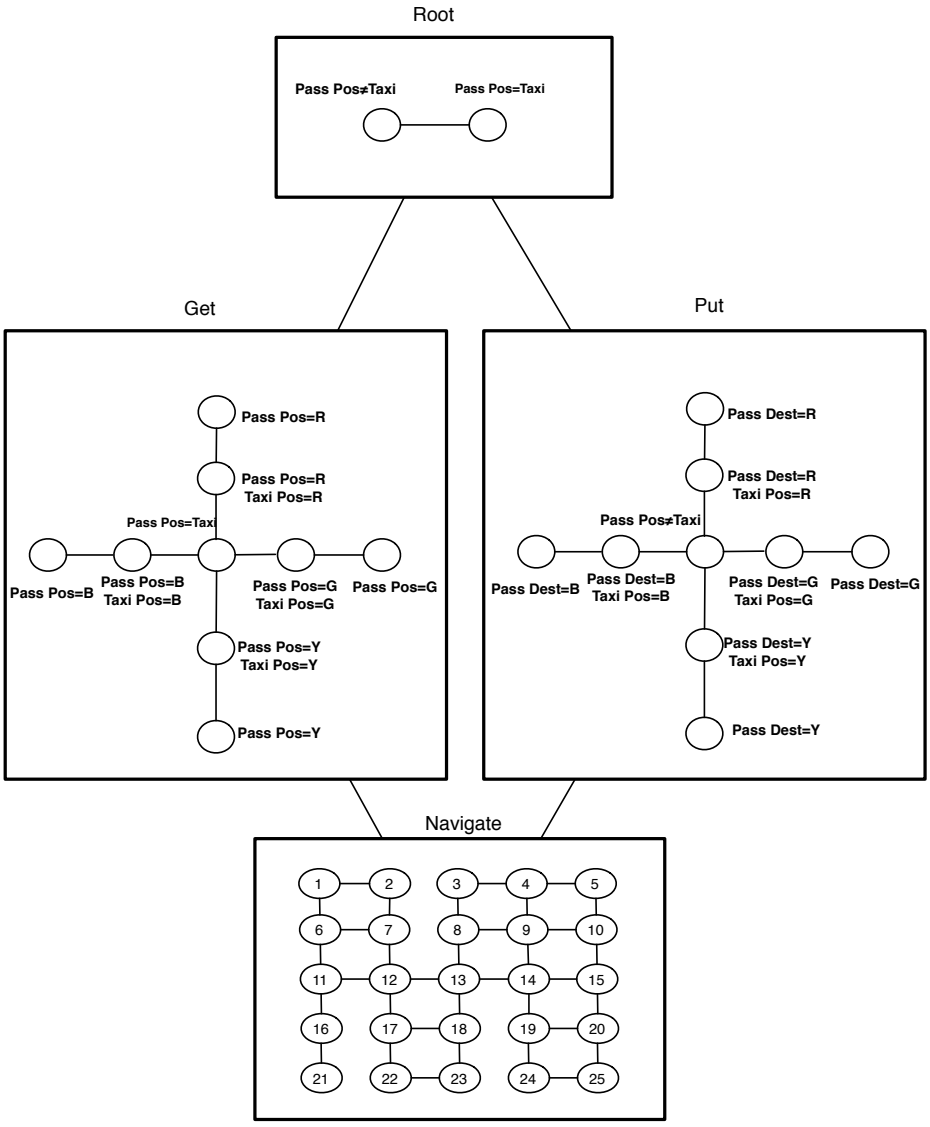


Figure 6.10. The reduced graphs for the taxi task.

a given subtask i, we define φ_χ as the compression given by the abstraction function χ_i, φ_G is the compression created by the reduced graph, and φ_e is the compression of the eigenvectors of the graph Laplacian. The basis functions $\phi_i(s)$ for subtask i and a state s can be written as the concatenation of the local basis functions with the basis functions from the child subtasks:

$$\phi_i(s) = [\varphi_e(\varphi_G(\varphi_\chi(s))) \mid \forall a \in A_i(s)\phi_a(s)],$$

where $a \in A_i(s)$ is a subtask, not a primitive action. Since our approach uses reward independent basis functions, the basis functions from parameterized tasks are only used once.

This approach allows methods, such as graph Laplacian basis functions, to be scaled to larger domains. The reduced graph can greatly reduce the size of the eigen problem that must be solved to create these basis functions.

In the introduction, we gave a generic description of how basis function decomposition might occur for the taxi task. Figure 6.11 shows the actual decomposition of our recursive basis function construction approach. For a subtask i the basis functions $\phi_i(s)$ are composed of two parts: $\bar{\phi}_{G_i}(s)$ are the basis functions constructed from the reduced graph G_i and $\phi_a(s)$ the basis functions from all of the unique child subtasks $a \in A_i(s)$. For example, the basis functions for the *get* task are constructed from the basis functions from the reduced graph G_{get} and the basis functions from the *navigate* subtask. For the *root* subtask, the basis functions are constructed from the reduced graph G_{root}, the basis functions from the *get* subtask, and the basis functions from the *put* subtask. The basis functions from the *navigate* subtask could potentially be used twice by the *navigate* subtask. Both the *get* and *put* subtasks construct their basis functions using the basis functions from the *navigate* subtask. However, the "extra" set of *navigate* basis functions provide no additional information and can be omitted.

In the previous chapters, we demonstrated that constructing basis functions directly in state-action space can significantly speed up learning. Since actions at a lower level are not available at a higher level, recursively generating state-action basis functions is not necessarily straightforward. Thus, our recursive basis function approach constructs basis functions over the state space.

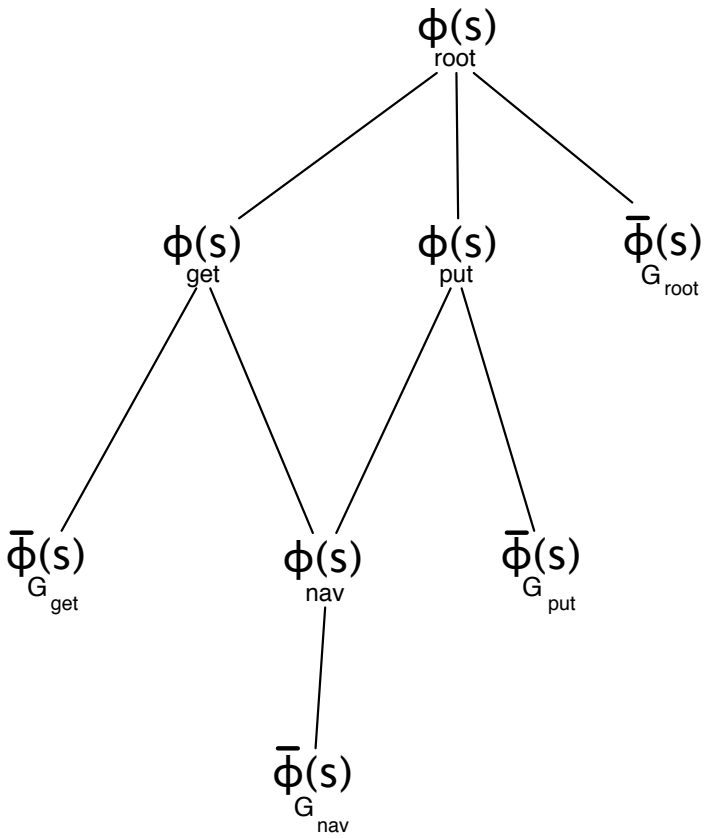


Figure 6.11. The recursive basis function decomposition from our proposed approach.

6.3 Analysis

In this section, we analyze our approach to basis function construction for HRL. We start by examining the abstractions created using the reduced graph approach. We demonstrate that the reduced graph approach is capable of finding abstractions similar to three types of abstraction outlined by Dietterich (2000).

The first type of abstraction involves eliminating state variables that are irrelevant to a subtask. We call this *subtask irrelevance*.

Definition 6.6 Subtask Irrelevance: *Given an MDP M with a state-abstracted task hierarchy H, a set of state variables Y_i are irrelevant to subtask M_i, if the state variables of M can be partitioned into two sets X_i and Y_i. The set of state variables Y_i are irrelevant if, for*

any stationary abstract hierarchical policy π that can be executed by i and its descendants, the following properties hold:

- *at subtask i the transition probability distribution $P_i^\pi(s', N|s, a)$ can be factored into the product of two distributions $P_i^\pi(\mathbf{x}', \mathbf{y}', N|\mathbf{x}, \mathbf{y}, a) = P_i^\pi(\mathbf{y}'|\mathbf{x}, \mathbf{y}, a) \cdot P_i^\pi(\mathbf{x}', N|\mathbf{x}, a)$*
- *$V_a^\pi(s_1) = V_a^\pi(s_2)$ and $\tilde{R}_i(s_1) = \tilde{R}_i(s_2)$, for any child action a and any pair of states $s_1 = (\mathbf{x}_i, \mathbf{y}_1)$ and $s_2 = (\mathbf{x}_i, \mathbf{y}_2)$ such that $\chi_i(s_1) = \chi_i(s_2) = \mathbf{x}_i$.*

An intuitive way to think about this form of abstraction is if a set of state variables are entirely irrelevant to a subtask, they do not play a role in the transition probability function and the reward function for subtask i.

The reduced graph construction algorithm constructs a reduced graph containing this abstraction. If the state variables in Y_i have no bearing on the probability transition function, they will be irrelevant in terms of connectivity on the graph and only X_i will be used to represent the state variables.

The second type of abstraction results from the structure of the hierarchy. Dietterich (2000) refers to this as shielding.

Definition 6.7 Shielding: *The value of s does not need to be represented for a subtask M_i, if for all paths from the root of the hierarchy H to subtask i there is some subtask j whose termination predicate $\beta_j(s)$ is true.*

Our approach can automatically find this representation because the graph is constructed over states in the set of samples $\mathcal{D}_i$ collected during the agent's initial exploratory period. $\beta_j(s)$ causes j to terminate and j lies on all paths between subtask i and the root. Thus, $\mathcal{D}_i$ cannot contain s, because the agent cannot transition to s during the execution of this subtask. Therefore, the graph will not include s, and s will not be represented in the basis functions.

The third type of abstraction results from "funnel actions," specifically the result distribution irrelevance condition (Dietterich, 2000).

Definition 6.8 Result distribution irrelevance: *For a given subtask i, result distribution irrelevance constructs an abstraction for all pairs of state s_1 and s_2, where the state variables can be partitioned into two sets $\{X_j, Y_j\}$, such that s_1 and s_2 only differ in their values of Y_j. The completion function for subtask i can be represented as an abstract completion function $C_i^\pi(\mathbf{x}_j, j)$, if the subset of state variables Y_j are irrelevant for the result distribution of child subtask j. Y_j is irrelevant for the result distribution of subtask j, if $P^\pi(s', N|s_1, j) = P^\pi(s', N|s_2, j), \forall s'$ and N.*

Result distribution irrelevance is an abstraction over state-action pairs. The graph reduction algorithm creates an abstract state for states s_1 and s_2 when $A_i(s_1) = A_i(s_2)$ and the state variables Y_i are irrelevant to connectivity of s to next state vertices s' for all $a \in A_i(s_1)$.

The abstraction created by the reduced graph is more strict than that described in result distribution irrelevance because it requires the constraint to be true for all available actions. However, it does not require the probabilities to be identical, just the connectivity within the graph.

In general, abstractions formed due to the graph reduction algorithm are no longer "safe" state abstractions. The graph reduction algorithm does not use probabilities to construct the abstractions but instead uses connectivity within the graph. This may lead to abstractions that "overgeneralize." For example, if $P^\pi(s', N|s_1, j)$ is slightly different than $P^\pi(s', N|s_2, j)$ but both values are greater than zero, then both s_1 and s_2 could potentially be collapsed into the same abstract vertex. Additionally, the reductions created by graph reduction algorithm construct "funnel action" abstractions, which Dietterich (2000) shows to be unsafe in the discounted reward setting. However, information is regained when basis functions from child subtasks are used in constructing basis functions. This information regains some of the "lost" information and allows the agent to learn appropriate policies.

6.4 Experimental Analysis

In the previous section, we analyzed the types of compressions generated by our approach. In this section, we experimentally evaluate the approach and compare them to other techniques.

6.4.1 Taxi

We evaluated four different techniques on the taxi task: hierarchical recursive graph Laplacian basis functions, graph Laplacian basis functions using the more traditional approach, RBFs, and table-lookup on the Taxi task. The results can be seen in Figure 6.12. The results of each experiment are averaged over 30 trials. The results plot the average number of primitive actions taken in the domain by the average cumulative reward received by the agent.

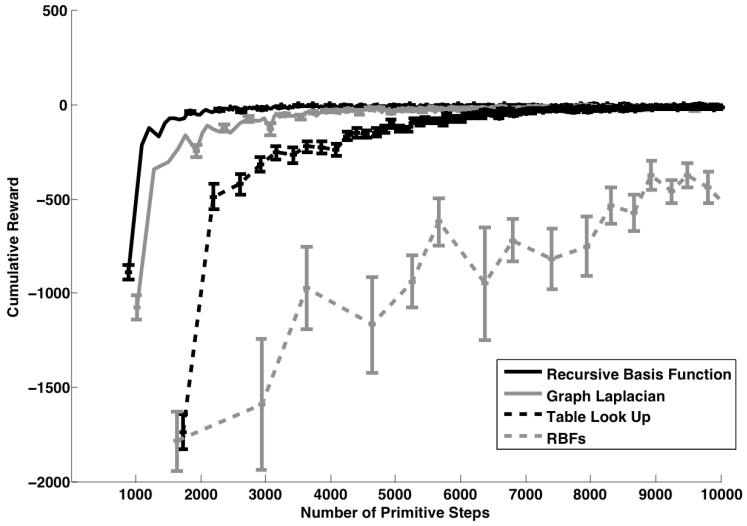


Figure 6.12. Results for the Taxi domain

The function approximation techniques all use a similar number of basis functions. Our results use the directed normalized graph Laplacian. The recursive basis function approach calculated ten local basis functions for the *navigate* subtask, nine basis functions

for *get*, and seven basis functions for *put*. The basis functions of the graph Laplacian approach were created by using the eigenvectors of the directed graph Laplacian of the state-abstracted graph. Ten basis functions were used for all of the subtasks. It is important to note that while a similar number of basis functions were used for both of the graph based approaches the amount of effort required to calculate the eigenvectors of the reduced graph is significantly less because the reduced graph is smaller than the state-abstracted graph. The recursive approach also uses basis functions from lower levels in order to obtain a better approximation.

The *navigate* subtask had a total of 17 basis functions created by uniformly placing the RBFs with two states between each RBF. The *get* and *put* subtasks had 21 basis functions created by placing the RBFs uniformly with five states between each RBF. We experimented with different numbers of RBFs but even doubling the number of basis functions did not greatly improve performance. Table 6.1 lists the number of basis functions used in the experiments for the taxi experiments.

	Navigate	Get	Pickup
Recursive Basis Function Construction	10 (local)	9 (local), 19 total	7 (local), 17 total
Graph Laplacian	10	10	10
Table Look Up	25	101	101
RBFs	17	21	21

Table 6.1. Number of basis functions used in the taxi experiments

6.4.2 Manufacturing Domain

We also evaluated our approach on a simulated manufacturing shown in Figure 6.13. This domain a modified version of the domain found in Ghavamzadeh and Mahadevan (2007). This domain models a manufacturing environment. The agent travels between the 33 locations. $M1-M3$ are workstations. The agent carries one part at a time to workstation drop off buffers $D1-D3$ and the assembled parts are brought from the workstation pick up

buffers, $P1 - P3$, to the warehouse. A reward of -5 is given when the actions Put1-Put3, Pick1-Pick3, Load1-Load3, and Unload actions are executed illegally. All other actions receive a reward of -1. The task is complete when the agent drops of one of each type of assembled part at the warehouse and a reward of 100 is given.

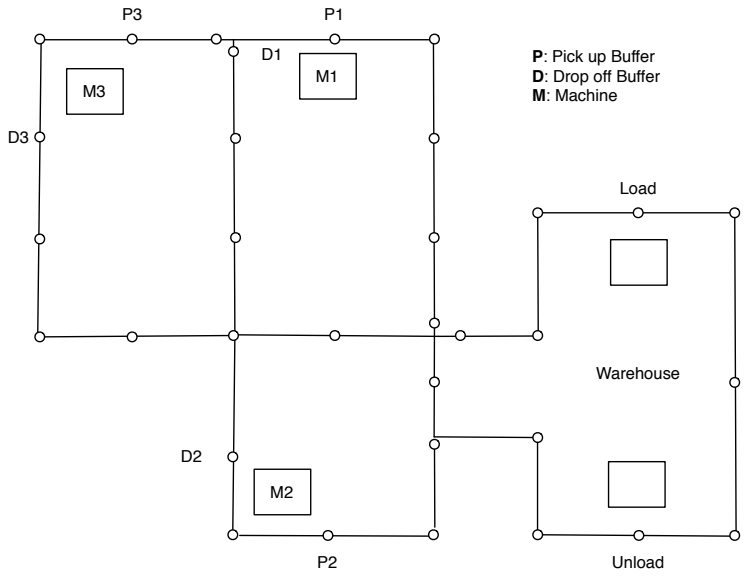


Figure 6.13. The Manufacturing Domain

The factored state consists of the number of parts in the pickup and drop off buffers, if the warehouse contains the three types of parts, the agent's location, the agent's status, and if each assembled part has been delivered. The flat representation of the state space consists of 33 locations, 6 buffers of size 2, 7 possible states of the agent, 2 values for each part in the loading area of the warehouse, and 2 values for each assembled part in the unloading area of the warehouse. This gives a total of $33 \times 3^6 \times 7 \times 2^3 \times 2^3 = 10,777,536$ states. There are 14 primitive actions: North, South, East, West, Put1-Put3, Pick1-Pick3, Load1-Load3, Unload, and Idle. The total number of parameters that must be learned in the flat case is $10,777,536 \times 14 = 161,663,040$.

Figure 6.14 defines a task hierarchy. The Navigate task moves the agent throughout the grid. DM1-DM3 tasks pickup the part from the warehouse and deliver it to the respective

machine. DA1-DA3 tasks pickup the assembled part from the correct machine and deliver it to the warehouse.

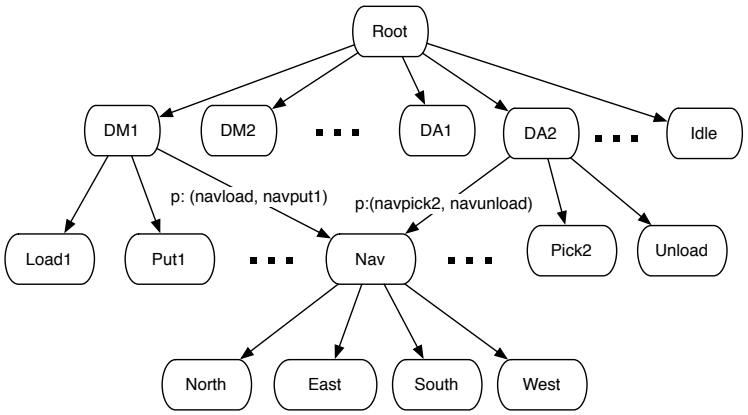


Figure 6.14. Hierarchy for the Manufacturing Domain

We evaluated the recursive basis function approach and compared it to table look up on this task. We cut off learning after 3000 primitive steps were taken in the domain. Our results use the normalized graph Laplacian. The recursive basis function approach created 15 local basis functions for the Navigate subtask, 10 basis functions for subtasks: DM1-DM3, and DA1 - DA3. The root subtask has 400 local basis functions. The results of learning can be seen in Figure 6.15. The results of each experiment was averaged over 30 trials.

6.4.3 Discussion of Results

Our results demonstrate that automatically constructing basis functions for hierarchical reinforcement learning significantly improve learning performance. Basis functions provide generalization over the state space of each subtask allowing the agent to learn about similar states.

Our recursive basis function construction approach has several advantages that help its performance. The reduced graph often has significantly fewer states and thus the agent

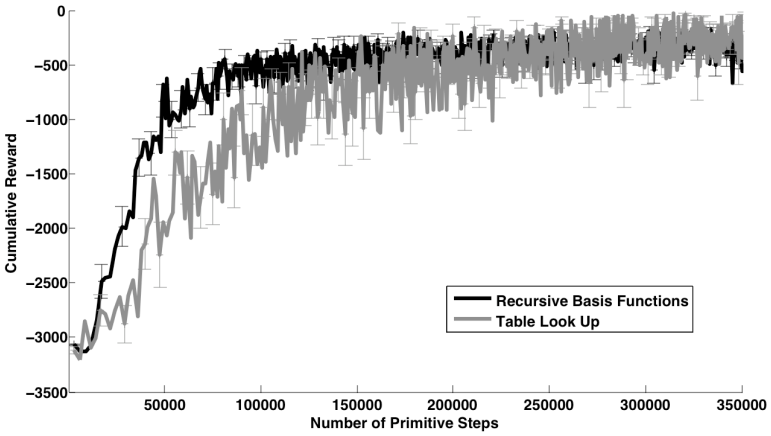


Figure 6.15. Results for the manufacturing domain

must learn fewer values. This is also beneficial when constructing the basis functions since the size of the eigen problem that must be solved is reduced.

The recursive approach is also helpful when the state space is not fully sampled during basis function construction and out of sample extension must be performed such as the Nyström extension (Williams & Seeger, 2001). Out of sample extension techniques perform best when there are states in the graph that are similar to the new previously unseen state. It also requires an accurate distance metric to link previously unseen states to states in the graph. One of the common properties of task hierarchies is that lower level subtasks are defined over a subset of the state variables. This means that while the agent may not have observed the state at a higher level subtask, lower level subtasks will have a representation for the state. Thus, even if out of sample extension for subtask i performs poorly the basis functions from lower level subtasks are likely to be accurate.

Another benefit of using the reduced graph is that basis functions for higher level subtasks are smoother. In Chapter 5 we noted that the smoothness of a function f can be affected by the invariant distribution Ψ. We observed that vertices with large values in Ψ will contribute more to the Sobolev norm and thus functions will be forced to be smooth in these states. Higher level subtasks often have a small number of states with large val-

ues in Ψ because child subtasks lead to a small subset of the state space that we will refer to as *funnel states*. In these subtasks, the eigenvectors of the graph Laplacian are forced to be smooth for the funnel states, which are a small subset of the states. This results in eigenvectors that are frequently delta functions even for low valued eigenvectors and thus a significant number of eigenvectors were required for learning. Eigenvectors created from the reduced graph are often significantly smoother since many of the vertices are merged. These smoother basis functions are more useful for approximating the value function.

6.5 Conclusion

In this chapter, we introduced an approach for automatic basis function construction when the agent has access to a task hierarchy. We discussed some of the issues that must be considered when automatically constructing basis functions for hierarchical reinforcement learning. We introduced an approach that constructs a graph over the abstract state space. We describe an algorithm to construct state abstractions based upon graph properties. Our approach constructs basis functions for lower level child subtasks as well as the eigenvectors of the reduced graph for the subtask.

We evaluated the performance of this approach experimentally and demonstrated that automatic basis function construction can significantly improve the speed of learning for traditional function approximation techniques as well as over exact methods.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

This dissertation demonstrates that leveraging information about the agent's action space during automatic basis function construction results in significantly improved performance. In this chapter, we provide a summary of the methods and algorithms presented in the dissertation, along with potential areas for future research.

7.1 Summary

In this dissertation, we investigated approaches for representation discovery in discrete Markov decision processes and discrete time SMDPs. Representation discovery is an area of vital importance for machine learning and artificial intelligence (Mahadevan, 2008). There are many approaches to solving Markov decision processes, such as linear programming (de Farias & Van Roy, 2003), policy iteration (Howard, 1960), value iteration (Puterman, 1994), and reinforcement learning (Sutton & Barto, 1998). All of these approaches employ function approximation techniques to scale to domains that have large or continuous state spaces. Most previous work on function approximation techniques employ hand-engineered basis functions. In this dissertation, we explored approaches to automatically constructing these basis functions and demonstrate that automatically constructed basis functions significantly outperform more traditional, hand-engineered approaches.

This dissertation specifically examined two problems: how to automatically build representations for action-value functions by explicitly incorporating actions into the representation and how representations can be automatically constructed for hierarchical reinforcement learning problems in a way that takes advantage of the action hierarchy.

Our approach to basis function construction extends recent work that builds basis functions on graphs induced by an MDP. This is a desirable approach because incorporating actions into the framework is straightforward, and the approach captures the underlying structure of the domain. We extended the spectral graph approach to basis function construction (Mahadevan & Maggioni, 2007).

This dissertation focused on two approaches to leveraging information about the agent's actions when constructing the representation. The first examined explicitly incorporating actions into the bases for MDPs and SMDPs using state-action graphs. The second introduced an approach for automatic basis function construction when the agent has access to a task hierarchy.

Chapter 4 described an approach to automatically constructing basis functions over state-action space. Our approach extends the work of Mahadevan and Maggioni (2007) in which basis functions were constructed using spectral analysis of the state graph induced by an MDP. Basis functions are constructed using spectral analysis of the state-action graph. This approach captures the underlying structure of the state-action space of the MDP. We described two approaches to constructing these graphs and evaluated this approach for MDPs with discrete state and action spaces. Our results demonstrate that basis functions created using the state-action graph significantly improve learning performance when compared to basis functions created over the state space. This is due to the fact that basis functions constructed over state-action space are able to simultaneously generalize over both states and actions.

Chapter 5 extended work on automatic basis function construction to SMDPs. We described how both state and state-action graphs can incorporate information about the temporally extended activities or macro-actions, and demonstrated our approach using the options framework, where the agent has access to options with a fixed predefined policy. We experimentally evaluated this approach for SMDPs with discrete state and action spaces. Our results demonstrate that basis functions constructed from state-action graphs

significantly improve learning performance when compared to basis functions created over the state-graph. Additionally, our results show that the best weightings for state-action graphs include temporal and likelihood information.

Chapter 6 investigated how hierarchical reinforcement learning (HRL) can be used to scale up automatic basis function construction. We extend automatic basis function construction approaches to multi-level task hierarchies. The key idea behind our approach is that basis function construction can exploit the value function decomposition defined by a fixed task hierarchy. Once again, spectral graph based techniques was used for basis function construction. We demonstrated how graph construction algorithms can leverage abstractions provided by the task hierarchy. We also introduced an approach to automatically construct abstractions based upon graph properties, through a graph reduction algorithm. Our approach decomposes basis functions for a subtask into two parts: the basis functions generated from the graph constructed from the agent's experience and the basis functions of the children subtasks. Our results show that using function approximation combined with HRL leads to a significant speed-up in learning.

7.2 Future Work

This area of research offers many interesting avenues for future research.

7.2.1 Representation Discovery Using State-Action Graphs

7.2.1.1 Extension of State-Action Graphs to Continuous Spaces

The algorithms presented in Chapters 4 and 5 focused on representation discovery for MDPs with discrete state and action spaces. One important area of future work is to extend state-action graph creation in domains with continuous state and discrete (or continuous) actions. Mahadevan et al. (2006) extend the graph Laplacian basis function approach to continuous spaces, and Johns and Mahadevan (2007) extended the directed graph Laplacian approach to continuous state spaces. Their approach uses k-nearest neighbors to create a

graph and then prunes edges that do not respect the "directionality" of the actions. A similar approach is likely to work well for creating state-action graphs over continuous state spaces. In order to extend this approach to state-action graph construction, k-nearest neighbors requires a distance metric over state-action space. This leads to a fundamental question that needs to be explored: how to define an appropriate distance metric over a continuous state-action space.

7.2.1.2 Action Representation using Alternative Feature Types

The techniques in Chapters 4 and 5 demonstrate the usefulness of incorporating actions into features when creating basis functions using spectral graph analysis. An interesting next step is to test the usefulness of incorporating actions when using other approaches to feature creation on graphs, such as diffusion wavelets or shortest path measures. While eigenvectors have been shown to be useful features for learning, they have some limitations. One limitation is that eigenvectors are defined over the entire graph, creating global features. Wavelet approaches provide techniques to create features at multiple temporal and spatial scales. Incorporating actions into this type of representation will allow us to analyze the joint space of states and actions at multiple scales.

7.2.1.3 Basis Function Construction for Other Action Value Functions

In this dissertation we primarily focused our attention to approximating Q-value functions; however, there are other forms of action-value functions. Advantage functions (Chandrasekaran et al., 1997; Hall et al., 1998; Winkeler et al., 1999; Hall et al., 2000; Baird, 1993) are another type of action value function. Advantage updating was proposed to cope with systems where the value of possible actions will not differ by a significant amount. It was specifically proposed for systems working in continuous time or for discrete time with small time steps.

Advantage functions store the value $A(s, a)$ which represents the degree to which the expected total discounted reward is increased by performing action a relative to the action

currently considered to be the best action. The update using advantage functions is defined as:

$$A(s,a) = \frac{Q(s,a) - \max_{a'} Q(s,a')}{\Delta t}.$$

This update requires that the maximum advantage in any state should converge to zero. When this occurs for every state the advantage function is normalized. Originally it was suggested that one learn both a value function and an advantage function. However, Baird (1995) later demonstrated this is not necessary as the formula can be modified to not require both functions:

$$\begin{aligned} A_{t+1}(s,a) &= (1-\alpha)A_t(s,a) + \\ &\quad \alpha[\frac{1}{\Delta t}(r + \gamma^{\Delta t} \max_{a' \in A(s')} A_t(s',a')) + (1 - \frac{1}{\Delta t}) \max_{a^* \in A(s)} A_t(s,a^*)]. \end{aligned}$$

The advantage function will not approach zero for all actions. Thus, the chance of representing this function accurately with function approximation is increased. Since our approach is particularly interested in distinguishing between actions it may be well suited to advantage functions.

Another type of action-value function is the average reward function. Schwartz (1993) proposed R-learning, an average-reward RL technique. This technique stores $R(s,a)$ which represents the average adjusted value of performing a in state s.

The update is defined as

$$R_{t+1}(s,a) = R_t(s,a)(1-\beta) + \beta(r - \rho_t + \max_{a' \in A(s')} R_t(s',a')$$

$$\rho_{t+1} = \rho_t(1-\alpha) + \alpha[r + \max_{a' \in A(s')} R_t(s',a') - \max_{a^* \in A(s)} R_t(s,a^*)].$$

Using the different action-value functions will give insight into this approach's effectiveness approximating different types of functions.

7.2.2 Representation Discovery for Multi-Level Task Hierarchies

7.2.2.1 Extension to State-Action Space

The technique described in Chapter 6 constructs basis functions over the state space. Chapters 4 and 5 show that basis functions constructed over the state-action space significantly improve learning performance. In order to extend the recursive basis function approach to state-action space, an approach to recursively select basis functions from child subtasks for a particular action must be resolved.

7.2.2.2 Multi-Scale Representations for Hierarchical Reinforcement Learning

The latter portion of the dissertation focused on constructing basis functions for HRL. In this work, we assumed the task hierarchy was given to the agent. A substantial amount of work has been done on skill learning (Thrun & Schwartz, 1995; McGovern, 2001; Hengst, 2002; Şimşek & Barto, 2004; Bonarini et al., 2006; Mehta et al., 2008; Şimşek & Barto, 2008; Konidaris & Barto, 2009; Zang et al., 2009; Neumann et al., 2009). An interesting avenue of research is how features created during representation discovery may be useful for subtask creation.

Muli-scale representations such as diffusion wavelets (Mahadevan & Maggioni, 2006; Maggioni & Mahadevan, 2006a) and multigrid approaches (Ziv, 2004; Ziv & Shimkin, 2005) are approaches that automatically construct a hierarchy of basis functions. These approaches may lead to new insights and new types of skill learning.

7.2.3 Theoretical Analysis of Basis Function Construction

In this dissertation, we introduced new approaches to basis function construction. We evaluated our approach experimentally and provided some intuition about why these approaches should be expected to work. However, a more rigorous theoretical analysis of

this work is required. One question of interest is to derive convergence bounds for learning algorithms using basis functions constructed from the agent's experience. Another is to provide analysis of the approach introduced for basis function construction for hierarchical reinforcement learning in Chapter 6.

7.2.4 Extension to Partially Observable Markov Decision Processes

This dissertation has focused on situations where the agent has had full access to its state. However, in many real world domains the agent will only have access to partial information. Partially observable Markov decision processes (POMDPs) (Sondik, 1971; Kaelbling et al., 1998) are a generalization of MDPs. POMDPs assume that the sequential decision process can be modeled as an MDP but have the additional constraint that the agent cannot directly observe the state space of the underlying MDP. The agent must learn from local observations. POMDPs are often intractable to solve. The construction of appropriate basis functions may make learning more tractable. There has been some investigation of dimensionality reduction in POMDPs such as Poupart and Boutilier (2002); Roy and Gordon (2003); Li et al. (2007). Extending the approaches discussed in this dissertation may provide additional insight into value function approximation for POMDPs.

7.2.5 Incremental Basis Function Construction

Currently, much of the work on automatic feature creation generally assumes the agent has explored a task and has access to a representative set of samples. An incremental approach to feature creation is necessary in settings where an agent must begin learning immediately, because experience in the domain is expensive. Since the eigen decomposition of the full matrix would no longer be required, an incremental approach should also help scale spectral approaches, which are computationally expensive.

A naïve approach would recompute the eigenfunctions each time the graph is changed. However, if the graph was frequently being updated this would be quite expensive. Since the graph's overall topology should not drastically change when a small number of nodes

are added, it should be possible to update the eigenvectors without performing the full eigenvalue decomposition. The problem of incremental eigenvector updates has been examined in the context of PCA (Artac et al., 2002). Law and Jain (2006) propose a technique to perform incremental ISOMAP. In the paper they state that their technique extends to techniques using eigenvectors such as Laplacian eigenmap. Kempe and McSherry (2008) describe a decentralized approach for eigenvector computation when the nodes of the graph only know their neighbors. This approach could potentially be modified to construct features incrementally.

7.3 Closing Remarks

In this dissertation, we explored automatic basis function construction in discrete MDPs and SMDPs. We developed approaches for basis function construction that incorporate information about the actions into the representation. This dissertation introduces an approach to automatically construct basis functions over the state-action space of an MDP using a state-action graph. We also extended this approach to SMDPs. Our research demonstrates that basis functions constructed from state-action graphs significantly improve learning performance. Additionally, we describe how automatic basis function approaches can be scaled up by leveraging multi-level task hierarchies. Our research shows that task hierarchies can be used to scale automatic basis function construction to large tasks, and that the use of these basis functions significantly improves learning performance in hierarchical reinforcement learning problems.

Constructing these representations often requires a nontrivial amount of experience. We envision these representations being used during the lifelong learning of an agent. The agent will simultaneously build a repertoire of representations and skills based upon its experience. When faced with a new learning scenario, the agent can use the representations it has already constructed to solve the task or leverage them to construct new representations.

In this chapter, we outlined a few directions of future research that are related to the methods presented in this dissertation. Naturally, there many other questions that must be answered before these approaches can be used to effectively represent large complex systems. Our hope is that these methods will aid in the creation of techniques that will solve large complex problems that require learning from experience.

BIBLIOGRAPHY

Albus, J. S. (1981). *Brain, behavior, and robotics*. Byte Books.

Amarel, S. (1968). On representations of problems of reasoning about actions. *Machine Intelligence 3*, *3*, 131–171.

Artac, M., Jogan, M., & Leonardis, A. (2002). Mobile robot localization using an incremental eigenspace model. *Proceedings of the 2002 IEEE International Conference on Robotics and Automation* (pp. 1025–1030).

Baird, L. C. (1993). *Advantage updating* (Technical Report WL-TR-93-1146). Wright-Patterson Air Force Base Ohio: Wright Laboratory.

Baird, L. C. (1995). Residual algorithms: Reinforcement learning with function approximation. *International Conference on Machine Learning* (pp. 30–37).

Barto, A., & Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Special Issue on Reinforcement Learning, Discrete Event Systems Jouranl*, *13*, 41–77.

Belkin, M., Matveeva, I., & Niyogi, P. (2004). Regularization and semi-supervised learning on large graphs. *Proceedings of the International Conference on Computational Learning Theory (COLT)* (pp. 624–638).

Belkin, M., & Niyogi, P. (2001). Laplacian eigenmaps and spectral techniques for embedding and clustering. *Advances in Neural Information Processing Systems 14* (pp. 585–591). Cambridge, MA: MIT Press.

Belkin, M., & Niyogi, P. (2004). Semi-supervised learning on riemannian manifolds. *Machine Learning*, *56*, 209–239.

Bellman, R., & Dreyfus, S. E. (1959). Functional approximations and dynamic programming. *Math Tables and Other Aides to Computation*, *13*, 247–251.

Bjorck, A., & Golub, G. (1973). Numerical methods for computing angles between linear subspaces. *Mathematics of Computation*, *27*, 579–594.

Bonarini, A., Lazaric, A., Restelli, M., & Vitali, P. (2006). Self-development framework for reinforcement learning agents. *Proceedings of the Fifth International Conference on Development and Learning*.

Bowling, M., Ghodsi, A., & Wilkinson, D. (2005). Action respecting embedding. *International Conference on Machine Learning*.

Boyan, J. A. (1999). Least-squares temporal difference learning. *Proceedings of the 16th International Conference on Machine Learning* (pp. 49–56). San Francisco, CA: Morgan Kaufmann.

Bradtke, S., & Barto, A. (1996). Linear least-square algorithms for temporal difference learning. *Machine Learning*, *22*, 33–57.

Castañon, D. A., & Bertsekas, D. P. (1989). Adaptive aggregation methods for infinite horizon dynamic programming. *IEEE Transactions on Automatic Control*, *34*, 589–598.

Chandrasekaran, S., Manjunath, B. S., Wang, Y. F., Winkeler, J., & Zhang, H. (1997). An eigenspace update algorithm for image analysis. *Graphical Models and Image Processing*, *59*, 321–332.

Chapman, D., & Kaelbling, L. P. (1991). Input generalization in delayed reinforcement learning. *Proceedings of the 12th International Joint Conference on Artificial Systems* (pp. 726–731).

Chung, F. (1997). *Spectral Graph Theory*. Number 92 in CBMS Regional Conference Series in Mathematics. American Mathematical Society.

Chung, F. (2005). Laplacians and the Cheeger inequality for directed graphs. *Annals of Combinatorics*, *9*, 1–19.

Ciocarlie, M., Goldfeder, C., & Allen, P. (2007). Dimensionality reduction for hand-independent dexterous robotic grasping. *IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 3270–3275).

Dayan, P. (1993). Improving generalisation for temporal difference learning: The successor representation. *Neural Computation*, *5*, 613–624.

de Farias, D., & Van Roy, B. (2003). The linear programming approach to approximate dynamic programming. *Operations Research*, *15*, 850–856.

Dean, T., Givan, R., & Leach, S. M. (1997). Model reduction techniques for computing approximately optimal solutions for Markov decision processes. *Thirteenth Conference on Uncertainty in Artificial Intelligence (UAI-97)* (pp. 124–131).

Dhillon, I. (2001). Co-clustering documents and words using spectral graph partitioning. *Knowledge Discovery and Dating Mining Conference*.

Dietterich, T. (1998). The MAXQ method for hierarchical reinforcement learning. *In Machine Learning: Proceedings of the Fifteenth International* (pp. 118–126). Morgan Kaufman.

Dietterich, T. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, *13*, 277–303.

Dietterich, T., & Wang, X. (2002). Batch value function approximation via support vectors. *Proceedings of Neural Information Processing Systems (NIPS)*. MIT Press.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*, 269–271.

Driessens, K., Ramon, J., & Gärtner, T. (2006). Graph kernels and gaussian processes for relational reinforcement learning. *Machine Learning*, *64*, 91–119.

Drummond, C. (2002). Accelerating reinforcement learning by composing solutions of automatically identified subtasks. *Journal of Artificial Intelligence Research*, *16*, 59–104.

Engel, Y., Mannor, S., & Meir, R. (2003). Bayes meets Bellman: The gaussian process approach to temporal difference learning. *Proceedings of the 20th International Conference on Machine Learning*.

Farley, B. G., & Clark, W. A. (1954). Simulation of self-organizing systems by digital computer. *IRE Transactions on Information Theory*, *4*, 76–84.

Ferns, N., Panangaden, P., & Precup, D. (2004). Metrics for finite Markov decision processes. *Proceedings of the Twentieth Conference on Uncertainty in Artificial Intelligence* (pp. 162–169).

Ferris, B., Fox, D., & Lawrence, N. (2007). Wifi-slam using gaussian process latent variable models. *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)* (pp. 2480–2485).

Fodor, I. K. (2002). *A survey of dimension reduction techniques* (Technical Report). Lawrence Livermore National Lab, Livermore, CA.

Foster, D., & Dayan, P. (2002). Structure in the space of value functions. *Machine Learning*, 325–346.

Gärtner, T., Driessens, K., & Ramon, J. (2003). Graph kernels and gaussian processes for relational reinforcement learning. *Inductive Logic Programming, 13th International Conference, ILP 2003, Proceedings* (pp. 146–163). Springer.

Ghavamzadeh, M., & Mahadevan, S. (2007). Hierarchical average-reward reinforcement learning. *Journal of Machine Learning Research*, *8*, 2629–2669.

Giunchiglia, F., & Walsh, T. (1992). A theory of abstraction. *Artificial Intelligence*, *57*, 323–390.

Givan, R., Dean, T., & Greig, M. (2003). Equivalence notions and model minimization in markov decision processes. *Artificial Intelligence*, *147*, 163–223.

Givan, R., Leach, S. M., & Dean, T. (2000). Bounded-parameter markov decision processes. *Artificial Intelligence*, *122*, 71–109.

Goldstein, H., Poole, C., & Safko, J. (2002). *Classical mechanics*. Addison-Wesley.

Golub, G., & Loan, C. V. (1989). *Matrix computations*. Baltimore, MD: John Hopkins University Press.

Gonzalez, J., Rojas, I., Ortega, J., Pomares, J., Fernandez, J., & Diaz, A. F. (2003). , multiobjective evolutionary optimization of the size, shape, and position parameters of radial basis function networks for function approximation. *IEEE Transactions on Neural Networks*, *14*, 1478–1495.

Gorban, A. N., Kégl, B., Wunsch, D. C., & Zinovyev, A. (Eds.). (2007). *Principal manifolds for data visualization and dimension reduction*. Lecture Notes in Computational Science and Engineering. Springer.

Gordon, G. J. (1995). Stable function approximation in dynamic programming. *Proceedings of the 12th International Conference on Machine Learning* (pp. 261–268).

Grudic, G., & Mulligan, J. (2005). Topological mapping with multiple visual manifolds. *Proceedings of Robotics: Science and Systems*. Cambridge, USA.

Guestrin, C., Koller, D., Parr, R., & Venkataraman, S. (2003). Efficient solution algorithms for factored mdps. *Journal of AI Research*, *19*, 399–468.

Hall, P., Marshall, D., & Martin, R. (1998). Incremental eigenanalysis for classification. *British Machine Vision Conference* (pp. 286–295).

Hall, P., Marshall, D., & Martin, R. (2000). Merging and splitting eigenspace models. *IEEE Transactions of Pattern Analysis and Machine Intelligence*, *22*, 1042–1048.

Ham, J., Lin, Y., & Lee, D. D. (2005). Learning nonlinear appearance manifolds for robot localization. *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 1239–1244).

Haykin, S. (1999). *Neural networks - a comprehensive foundation*. New Jersey: Prentice Hall. 2nd edition.

Hengst, B. (2002). Discovering hierarchy in reinforcement learning with hexq. *Proceedings of the Nineteenth International Conference on Machine Learning* (pp. 243–250).

Howard, R. (1960). *Dynamic programing and markov decision processes*. MIT Press.

Huber, P. J. (1985). Projection pursuit. *Annals of Statistics*, *13*, 435–475.

Hyvvärinen, A. (1999). Survey on independent component analysis. *Neural Computing Surveys*, *2*, 94–128.

Jenkins, O. C., & Matarić, M. J. (2004). A spatio-temporal extension to isomap nonlinear dimensionality reduction. *Proceedings of the 21st International Conference on Machine Learning* (pp. 441–448).

Johns, J., & Mahadevan, S. (2007). Constructing basis functions from directed graphs for value function approximation. *Proceedings of Twenty-fourth International Conference on Machine Learning (ICML)*.

Jolliffe, T. (1986). *Principal components analysis*. Berlin: Springer.

Jong, N. K., & Stone, P. (2005). State abstraction discovery from irrelevant state variables. *Proceedings of the 19th International Joint Conference on Artificial Systems*.

Kaelbling, L. P., Littman, M., & Cassandra, A. H. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence Journal*, *101*, 99–134.

Karayiannis, N. B. (1999). Reformulated radial basis neural networks trained by gradient descent. *IEEE Transactions on Neural Networks*, *10*, 657–671.

Karni, Z., & Gotsmann, C. (2000). Spectral compression of mesh geometry. *SIGGRAPH '00: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques* (pp. 279–286). ACM Press/Addison-Wesley Publishing Co.

Keller, P. W., Mannor, S., & Precup, D. (2006). Automatic basis function construction for approximate dynamic programming and reinforcement learning. *Proceedings of the 23rd International Conference on Machine Learning*. New York, NY: ACM Press.

Kempe, D., & McSherry, F. (2008). A decentralized algorithm for spectral analysis. *Journal of Computer and System Sciences*, *74*, 70–83.

Kondor, R., & Vert, J.-P. (2004). *Kernel methods in computational biology*, chapter Diffusion, 171–192. MIT Press.

Konidaris, G., & Barto, A. (2009). Efficient skill learning using abstraction selection. *Proceedings of the Twenty First International Joint Conference on Artificial Intelligence*.

Konidaris, G., & Osentoski, S. (2008). *Value function approximation in reinforcement learning using the Fourier basis* (Technical Report UM-CS-2008-19). Department of Computer Science, University of Massachusetts Amherst.

Kretchmar, R. M., & Anderson, C. W. (1999). Using temporal neighborhoods to adapt function approximators in reinforcement learning. *International Work Conference on Artificial and Natural Neural Networks*.

Lagoudakis, M., & Parr, R. (2003). Least-squares policy iteration. *Journal of Machine Learning Research*, *4*, 1107–1149.

Law, M. H. C., & Jain, A. K. (2006). Incremental nonlinear dimensionality reduction. *IEEE Transactions of Pattern Analysis and Machine Intelligence*, *28*, 377–391.

Lazaro, M., Santamaria, I., & Pantaleon, C. (2003). A new EM-based training algorithm for RBF networks. *Neural Networks*, *16*, 69–77.

Li, L., Walsh, T. J., & Littman, M. (2006). Towards a unified theory of state abstraction for mdps. *Proceedings of the 23rd International Conference on Machine Learning*.

Li, X., Cheung, W. K. W., Liu, J., & Wu, Z. (2007). A novel orthogonal NMF-based belief compression for POMDPs. *Proceedings of the 24th International Conference on Machine Learning*.

Maggioni, M., & Mahadevan, S. (2006a). *A multiscale framework for Markov decision processes using diffusion wavelets* (Technical Report TR-2006-36). University Of Massachusetts, Department of Computer Science.

Maggioni, M., & Mahadevan, S. (2006b). *A multiscale framework for Markov decision processes using diffusion wavelets* (Technical Report TR-2006-36). University of Massachusetts, Department of Computer Science.

Mahadevan, S. (2005). Proto-Value Functions: Developmental Reinforcement Learning. *Proceedings of the 22nd International Conference on Machine Learning* (pp. 553–560). New York, NY: ACM Press.

Mahadevan, S. (2008). *Representation discovery using harmonic analysis*. Morgan Claypool Publishers.

Mahadevan, S. (2009). The learning of representation and control in Markov decision processes: New frontiers. *Foundations and Trends in Machine Learning*.

Mahadevan, S., & Maggioni, M. (2006). Value function approximation using diffusion wavelets and Laplacian eigenfunctions. *Neural Information Processing Systems*. MIT Press.

Mahadevan, S., & Maggioni, M. (2007). Proto-value functions: A laplacian framework for learning representation and control in Markov decision processes. *Journal of Machine Learning*, *8*, 2169–2231.

Mahadevan, S., Maggioni, M., Ferguson, K., & Osentoski, S. (2006). Learning representation and control in continuous Markov decision processes. *Proceedings of the 21st National Conference on Artificial Intelligence*. Menlo Park, CA: AAAI Press.

Mardia, K. V., Kent, J. T., & Bibby, J. M. (1995). *Multivariate analysis*. Academic Press.

McCallum, A. (1995). *Reinforcement learning with selective perception and hidden state*. Doctoral dissertation, University of Rochester.

McGovern, A. (2001). *Autonomous discovery of temporal abstractions from interaction with an environment*. Doctoral dissertation, University of Massachusetts, Amherst.

McLoone, S., Brown, M. D., Irwin, G., & Lightbody, G. (1998). A hybrid linear/nonlinear training algorithm for feedforward neural networks. *IEEE Transactions on Neural Networks*, *9*, 669–684.

Mehta, N., Ray, S., Tadepalli, P., & Dietterich, T. (2008). Automatic discovery and transfer of MAXQ hierarchies. *Proceedings of the 25th International Conference on Machine Learning* (pp. 648–655). Helsinki, Finland.

Menache, I., Mannor, S., & Shimkin, N. (2002). Q-cut - dynamic discovery of sub-goals in reinforcement learning. *Proceedings of the Thirteenth European Conference on Machine Learning* (pp. 295–306).

Menache, I., Mannor, S., & Shimkin, N. (2005). Basis function adaptation in temporal difference reinforcement learning. *Annals of Operations Research*, *134*, 215–238.

Meyer, C. (1989). Uncoupling the perron eigenvector problem. *Linear Algebra and its Applications*, *114/115*, 69–94.

Moody, J., & Darken, C. (1989). Fast learning in networks of locally tuned processing units. *Neural Computation*, *1*, 281–294.

Nedic, A., & Bertsekas, D. P. (2003). Least-squares policy evaluation algorithms with linear function approximation. *Discrete Event Systems Journal*, 13.

Neumann, G., Maass, W., & Peters, J. (2009). Learning complex motions by sequencing simpler templates. *Proceedings of the 26th International Conference on Machine Learning*.

Ng, A., Jordan, M., & Weiss, Y. (2002). On spectral clustering: Analysis and an algorithm. *Proceedings of the Neural Information Processing Systems*.

Olson, E., Walter, M., Teller, S., & Leonard, J. (2005). Single-cluster spectral graph partitioning for robotics applications. *Robotics: Science and Systems*.

Ormoneit, D., & Sen, S. (2002). Kernel-based reinforcement learning. *Machine Learning*, *49*, 168–178.

Page, L., Brin, S., Motwani, R., & Winograd, T. (1998). *The PageRank citation ranking: Bringing order to the web* (Technical Report). Stanford University.

Parr, R., Painter-Wakefield, C., Li, L., & Littman, M. (2007). Analyzing feature generation for value-function approximation. *Proceedings of the International Conference on Machine Learning (ICML)*.

Parr, R., & Russell, S. (1998). Reinforcement learning with hierarchies of machines. *Advances in Neural Information Processing Systems 10* (pp. 1043–1049). MIT Press.

Petrik, M. (2007). An analysis of Laplacian methods for value function approximation in mdps. *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

Poupart, P., & Boutilier, C. (2002). Value-directed compression of POMDPs. *Advances in Neural Information Processing Systems 15 (NIPS)* (pp. 1547–1554).

Powell, M. J. D. (1987). Radial basis functions for multivariate interpolation: A review. In *Algorithms for approximation*. Oxford: Clarendon Press.

Precup, D., Sutton, R., & Singh, S. (2000). Eligibility traces for off-policy policy evaluation. *Proceedings of the 17th International Conference on Machine Learning* (pp. 759–766). Morgan Kaufmann.

Puterman, M. L. (1994). *Markov decision processes*. New YorkTemporal, USA: Wiley Interscience.

Rasmussen, C. E., & Kuss, M. (2004). Gaussian processes in reinforcement learning. *Advances in Neural Information Processing Systems (NIPS) 16*.

Ratitch, B., & Precup, D. (2004). Sparse distributed memories for on-line value-based reinforcement learning. *Proceedings of the 15th European Conference on Machine Learning* (pp. 347–358).

Ravindran, B. (2004). *An algebraic approach to abstraction in reinforcement learning*. Doctoral dissertation, University of Massachusetts.

Ravindran, B., & Barto, A. (2003). SMDP homomorphisms: An algebraic approach to abstraction in semi Markov decision processes. *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 03)* (pp. 1011–1016). AAAI Press.

Rogers, D. F., Plante, R. D., Wong, R. T., & Evans, J. R. (1991). Aggregation and disaggregation techniques and methodology in optimization. *Operations Research*, *39*, 553–582.

Roweis, S. T., & Saul, L. K. (2000). Nonlinear dimensionality reduction by locally linear embedding. *Science*, *290*, 2323–2326.

Roy, N., & Gordon, G. J. (2003). Exponential family PCA for belief compression in POMDPs. *Advances in Neural Information Processing Systems*.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, *3*, 210–229.

Sanchez, D. (1995). Second derivative dependent placement of RBF centers. *Neurocomputing*, *7*, 311–317.

Schwartz, A. (1993). A reinforcement learning method for maximizing undiscounted rewards. *Proceedings of the Tenth International Conference on Machine Learning* (pp. 298–205).

Shannon, C. (1950). Programming a computer for playing chess. *Philosophical Magazine*, *41*, 256–275.

Shi, J., & Malik, J. (2000). Normalized cuts and image segmentation. *EEE Transactions on Pattern Analysis and Machine Intelligence*, *22*, 888–905.

Şimşek, Ö., & Barto, A. (2004). Using relative novelty to identify useful temporal abstractions in reinforcement learning. *Proceedings of the Twenty-First International Conference on Machine Learning* (pp. 751–758).

Şimşek, Ö., & Barto, A. (2008). Skill characterization based on betweenness. *Neural Information Processing Systems (NIPS)*.

Şimşek, Ö., Wolfe, A. P., & Barto, A. (2005). Identifying useful subgoals in reinforcement learning by local graph partitioning. *Proceedings of the Twenty-Second International Conference on Machine Learning (ICML-05)*.

Singh, S., Jaakkola, T., & Jordan, M. (1995). Reinforcement learning with soft state aggregation. *Advances in Neural Information Processing Systems 8 (NIPS)* (pp. 361–368). MIT Press.

Smart, W. (2004). Explicit manifold representations for value-function approximation in reinforcement learning. *Prceedings of the 8th International Symposium on Artificial Intelligence and mathematics*.

Smith, A. J. (2002). Applications of the self-organising map to reinforcement learning. *Neural Networks*, *15*, 1107–1124.

Sondik, E. J. (1971). *The optimal control of partially observable markov processes*. Doctoral dissertation, Stanford University.

Sugiyama, M., Hachiya, H., Towell, C., & Vijayakumar, S. (2007). Value function approximation on non-linear manifolds for robot motor control. *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'07)*. Rome, Italy.

Sugiyama, M., Hachiya, H., Towell, C., & Vijayakumar, S. (2008). Geodesic gaussian kernels for value function approximation. *Autonomous Robots*, *25*, 287–304.

Sutton, R., & Barto, A. (1998). *Reinforcement learning*. Cambridge, MA: MIT Press.

Sutton, R., Precup, D., & Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, *112*, 181–211.

Tenenbaum, J. B., de Silva, V., & Langford, J. (2000). A global geometric famework for nonlinear dimensionality reduction. *Science*, 2319.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, *8*, 257–278.

Thrun, S., & Schwartz, A. (1995). Finding structure in reinforcement learning. *Advances n Neural Information Processing Systems* (pp. 385–392).

Tsitsiklis, J., & Van Roy, B. (1997). An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, *42*, 674–690.

Tsoli, A., & Jenkins, O. C. (2007). 2D subspaces for user-driven robot grasping. *Robotics: Science and Systems - Robot Manipulation: Sensing and Adapting to the Real World*.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards*. Doctoral dissertation, Cambridge University.

Wedin, P. A. (1983). On angles between subspaces of a finite dimensional inner product space. In *Matrix pencils*, Lecture Notes In Mathematics 973, 263–285. Springer.

Williams, C. K. I., & Seeger, M. (2001). Using the nystrom method to speed up kernel machines. *Advances in Neural Information Processing Systems 13*.

Winkeler, J., Manjunath, B. S., & Chandrasekaran, S. (1999). Subset selection for active object recognition. *IEEE Conference on Computer Vision and Pattern Recognition* (pp. 511–512).

Wolfe, A. P., & Barto, A. (2006). Decision tree methods for finding reusable MDP homomorphisms. *Proceedings of the 21st International Conference on Artificial Intelligence*.

Yairi, T. (2007). Map building without localization by dimensionality reduction techniques. *Proceedings of the 24th International Conference on Machine Learning* (pp. 1071–1078).

Ye, N. (2003). *A comparative analysis of dimensionality reduction techniques*. Human Factors and Ergonomics Series. Lawrence Erlbaum.

Zang, P., Zhou, P., Minnen, D., & Isbell, C. (2009). Discovering options from example trajectories. *Proceedings of the 26th International Conference on Machine Learning*.

Ziv, O. (2004). Algebraic multigrid for reinforcement learning. Master's thesis, The Technion - Israel Institute of Technology.

Ziv, O., & Shimkin, N. (2005). Multigrid methods for policy evaluation and reinforcement learning. *International Symposium on Intelligent Control* (pp. 1391–1396).